Statistical Debugging for Simulations

ROSS GORE, Old Dominion University
PAUL F. REYNOLDS JR., University of Virginia
DAVID KAMENSKY, University of Texas at Austin
SAIKOU DIALLO and JOSE PADILLA, Old Dominion University

Predictions from simulations have entered the mainstream of public policy and decision-making practices. Unfortunately, methods for gaining insight into faulty simulations outputs have not kept pace. Ideally, an insight gathering method would automatically identify the cause of a faulty output and explain to the simulation developer how to correct it. In the field of software engineering, this challenge has been addressed for general-purpose software through statistical debuggers. We present two research contributions, *elastic predicates* and *many-valued labeling functions*, that enable debuggers designed for general-purpose software to become more effective for simulations employing random variates and continuous numbers. Elastic predicates address deficiencies of existing debuggers related to continuous numbers, whereas many-valued labeling functions support the use of random variates. When used in combinations, these contributions allow a simulation developer tasked with localizing the program statement causing the faulty simulation output to examine 40% fewer statements than the leading alternatives. Our evaluation shows that elastic predicates and many-valued labeling functions maintain their ability to reduce the number of program statements that need to be examined under the imperfect conditions that developers experience in practice.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Debugging Aids

General Terms: Design, Algorithms, Experimentation

Additional Key Words and Phrases: Modeling and simulation, automated debugging, fault localization

ACM Reference Format:
Ross Gore, Paul F. Reynolds Jr., David Kamensky, Saikou Diallo, and Jose Padilla. 2015. Statistical debugging for simulations. ACM Trans. Model. Comput. Simul. 25, 3, Article 16 (April 2015), 26 pages.
DOI: http://dx.doi.org/10.1145/2699722

1. INTRODUCTION

Proper modeling and simulation (M&S) practice requires ensuring that no bugs occur when a conceptual model is translated into a computer simulation [Law and Kelton 1991]. The process assumes that the model design and desired input/output pairs have been specified. Then, the simulation is constructed by employing either (1) a simulation engine (i.e., Arena, Simulink), (2) a package or library (i.e., MATLAB, Facsimile, Altera), or (3) a general-purpose programming language (i.e., Java, C/C++). Unfortunately, no matter which option a developer chooses, constructing a bug-free simulation

Authors' addresses: R. Gore, S. Diallo, and J. Padilla, 1030 University Blvd., Virginia Modeling Analysis and Simulation Center, Old Dominion University, Suffolk, VA 23435; emails: {rgore, sdiallo, jpadilla}@odu.edu; P. F. Reynolds Jr., 151 Engineer's Way, Dept. of Computer Science, University of Virginia, Charlottesville, VA 22904; email: reynolds@virginia.edu; D. Kamensky, Institute for Computational Engineering and Science, 201 East 24th St, Stop C0200, POB 4.102, Austin, Texas 78712-1229; email: kamensky@ices.utexas.edu.

DOI: http://dx.doi.org/10.1145/2699722

is not an atomic action. Inherently, the process requires iterations of testing and debugging [Dijkstra 1976].

Most previous research in this domain focused on testing—determining if a simulation is producing the intended output for a given input. In comparison, there has been relatively little research on simulation debugging, which is the process of locating the cause of a faulty outcome in a simulation and correcting it [Krahl 2005]. This discrepancy needs to be addressed; simulation debugging is largely manual, tedious, time intensive, and costly. Automating even a portion of the task would significantly reduce the time spent developing simulations and would improve efficiency.

Debuggers employing statistical, machine learning, and data mining techniques have been developed to meet this goal for general-purpose software [Liblit 2008]. This area of research is referred to as statistical debugging.

Statistical debuggers require a set of test inputs, corresponding execution traces, and a labeling of the execution traces as passing or failing. The execution traces typically reflect coverage of individual statements or reflect the truth values of inserted predicates. The debuggers assign *suspiciousness* scores to these program elements to measure the likelihood that a given statement contains a fault. Then, the program statements are ranked in descending order of suspiciousness and returned to the developer tasked with localizing the cause of the unexpected output. Given the ranked set of statements, the *Cost* metric measures the percentage of executed statements a developer must examine before encountering the faulty statement.

Although statistical debuggers are able to localize faults while incurring negligible *Cost* for general-purpose software, they are not tailored to simulations. Continuous numbers, which are common in most simulations, are ignored by existing statistical debuggers because of (1) their relative scarcity in general-purpose software (2) and the additional amount of space and time their analysis entails compared to their discrete counterparts [Liblit 2008]. Furthermore, the use of random variates in stochastic simulations defies the assumptions made by statistical debuggers that executions are repeatable outcomes [Cleve and Zeller 2005; Jeffrey et al. 2008]. These factors result in developers incurring significant *Cost* when employing existing statistical debuggers to localize faults in simulations with (1) continuous numbers and/or (2) random variates.

In this work, we propose a statistical debugger—Exploratory Simulation Prediction (ESP)—and evaluate the extent to which it reduces the *Cost* of localizing the statement causing an unexpected output within simulations employing continuous numbers or random variates. ESP is targeted toward simulation developers who work with general-purpose programming languages; however, it is applicable to all developers provided they have access to (1) the source code of their simulation and (2) a set of test inputs paired with specified outputs to judge the correctness of the simulation.

The goal of ESP is to efficiently localize a *source code statement* within the simulation causing the simulation to fail to produce the specified output for a given input. This goal is related to, but not the same as, simulation validation, verification, and testing (VV&T). In general, VV&T is concerned with identifying inputs where the simulation does not produce the specified output [Sargent 2013]. ESP complements VV&T by then localizing the source code statement causing the input(s) to fail to produce the specified output. We review the relationship between ESP and VV&T further in Section 7.

The first research contribution within ESP, *elastic predicates*, is developed with continuous numbers in mind. Most general-purpose applications are almost entirely composed of discrete numbers and Boolean conditions. The values that these data types take on are indicators where the sign distinguishes one outcome from another [Liblit 2008]. As a result, when existing statistical debuggers insert predicates into a subject program's source code, the predicates test if a variable value is greater than, less than, or equal to zero.

Although the approach is effective for discrete numbers and Boolean conditions, it fails to capture some faulty value ranges taken on by continuous numbers. Elastic predicates address this shortcoming by employing summary statistics such as the mean and standard deviation to cluster together similar faulty values assigned to variables. When combined with the predicates already employed in existing statistical debuggers, elastic predicates reduce the *Cost* incurred by a simulation developer localizing the statement, causing an unexpected output in the simulations featured in our evaluation. Furthermore, we show that elastic predicates can be computed efficiently in terms of space and time.

The second research contribution within ESP is the first analytical result concerning many-valued, as opposed to Boolean, labeling functions for statistical debugging execution traces. Simulations are frequently employed to explore domains where physical experimentation is impossible due to economic, moral, and ethical constraints. In these cases, the expected output for a given test input is not known and thus impossible to provide. Even if analytical solutions are used to provide the expected output, some simulations include random variates and thus variance in their output. This creates at least two difficult questions for developers tasked with localizing the statement causing an unexpected output in simulations: (1) does an output have to match the analytical solution exactly to be labeled as a passing execution trace? and (2) if not, how close must the output of an execution trace be to the analytical solution to be considered passing and to what extent?

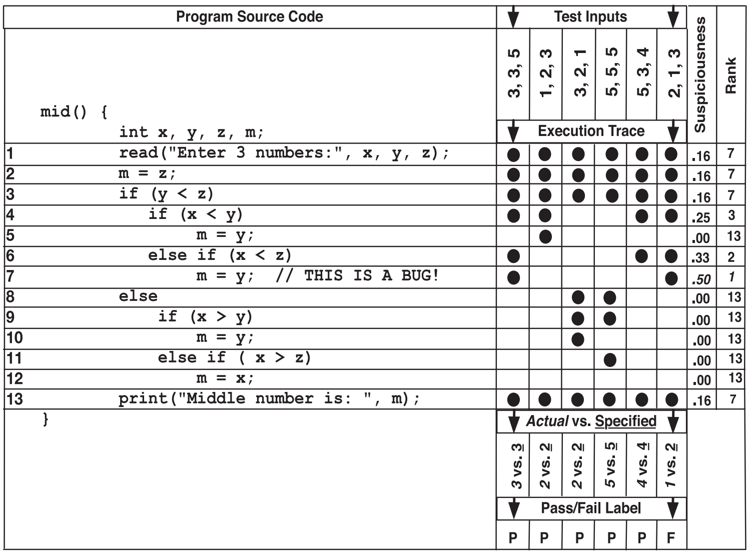
Many-valued labeling functions provide a framework to answer these questions. They enable a simulation developer to construct a function that specifies the extent to which the output of an execution trace passes and fails. This relaxes the statistical debugging requirement that execution traces must be labeled as strictly passing or failing. When employed in existing statistical debuggers, many-valued labeling functions reduce the *Cost* incurred by a simulation developer localizing the faulty statement, causing an unexpected output in the simulations featured in our evaluation.

Combined elastic predicates and many-valued labeling functions change the simulation debugging landscape. When featured in tandem, the two research contributions enable simulation developers to inspect nearly 40% fewer program statements (on average) than existing statistical debuggers for the 13 simulations with 69 different faulty versions included in our evaluation. Furthermore, these improvements are not only available in a perfect world, where (1) every statement is profiled all the time and (2) thousands of inputs exist for every simulation. Our evaluation shows that using sparse profiling and only 30 test inputs still enables simulation developers to inspect approximately 33% fewer program statements (on average) than existing statistical debuggers for the 13 simulations with 69 different faulty versions included in our evaluation.

The rest of the article is organized as follows. Section 2 provides a background of the inner workings of existing statistical debuggers, their utility, and deficiencies. Section 3 elaborates on the methodology behind elastic predicates and many-valued labeling functions. An extensive evaluation of the *Cost* incurred by developers employing elastic predicates and many-valued labeling functions compared to existing statistical debuggers is presented in Section 4, followed by an evaluation of ESP in an imperfect world in Section 5. A case study of ESP being employed in the wild and related work are discussed in Sections 6 and 7. Finally, Section 8 concludes our study.

2. STATISTICAL DEBUGGING BACKGROUND

Examples help elucidate the inner workings, utility, and deficiencies of existing statistical debuggers. Section 2.1 demonstrates how statistical debugging is effectively applied to isolate the fault in a small program used to compute the median of three



	Program Source Code	3, 3, 5	1, 2, 3	3, 2, 1	5, 5, 5	5, 3, 4	2, 1, 3	Suspiciousness	Rank
	`mid() {`								
	`int x, y, z, m;`	Execution Trace							
1	`read("Enter 3 numbers:", x, y, z);`	●	●	●	●	●	●	.16	7
2	`m = z;`	●	●	●	●	●	●	.16	7
3	`if (y < z)`	●	●	●	●	●	●	.16	7
4	`if (x < y)`	●	●			●	●	.25	3
5	`m = y;`		●					.00	13
6	`else if (x < z)`	●				●	●	.33	2
7	`m = y; // THIS IS A BUG!`	●					●	*.50*	*1*
8	`else`			●	●			.00	13
9	`if (x > y)`			●	●			.00	13
10	`m = y;`			●				.00	13
11	`else if ( x > z)`				●			.00	13
12	`m = x;`							.00	13
13	`print("Middle number is: ", m);`	●	●	●	●	●	●	.16	7
	`}`	*Actual* vs. Specified							
		3 vs. 3	*2* vs. 2	*2* vs. 2	*5* vs. 5	*4* vs. 4	*1* vs. 2		
		Pass/Fail Label							
		P	P	P	P	P	F		

Fig. 1. Statistical debugging example.

integers. In Section 2.2, a larger example uncovers the shortcomings of existing statistical debuggers. The background provided in these sections motivates and enables an understanding of our proposed simulation debugging improvements.

2.1. Inner Workings and Utility

Figure 1 shows the program, `mid()`, and its test suite. The program `mid()` takes three integers as input and is required to output the median value. It could be a function (1) programmed by a developer or (2) called by a library or package. Regardless of its origin, the function fails to properly identify the median number for some inputs because there is a fault in Statement 7. Statement 7 should read `m = x`; however, it reads `m = y`.

Figure 1 illustrates the process of employing statistical debugging to localize this fault. The debugger begins by executing `mid()` for each of the test inputs shown at the top of the figure. The execution of `mid` for each test input is traced to record the statements that are executed. The columns below the test inputs reflect each execution trace: a black dot signifies that the statement was executed, and the lack of a black dot signifies that the statement was not executed.

Once `mid()` is executed for a given test input, the actual output of the program is compared to the specified output. The actual and specified outputs for each test input are shown immediately below the execution trace. The actual output is written in italics and the specified output is underlined. These outputs determine if the corresponding execution trace is labeled as passing or failing. If the actual output matches the specified output, then the execution trace passes; otherwise, it fails. The result of applying this labeling process to each execution trace of `mid()` is shown in the bottom row of the figure.

Labeling each execution trace as passing or failing enables the suspiciousness and rank of each statement in the source code of `mid()` to be computed. Recall that suspiciousness measures how likely it is that a statement contains a fault. It is calculated by computing the ratio of the number of failing execution traces that include the statement to the number of total execution traces that include the statement. The suspiciousness of each statement is shown in second right-most column of Figure 1.

The right-most column of Figure 1 shows the rank of each statement. The ranking column reflects the sorted order of the statements in `mid()` when they are returned to the developer. The rank is the maximum number of statements that would have to be examined by the developer. It is assumed that a developer examines all statements with the same suspiciousness together.

In this figure, Statement 7 is identified as the most suspicious statement (.50) because it is only included in two execution traces: one that passes and one that fails. Every other statement is included in at least two passing execution traces, and no statement is included in more than one failing execution trace. As a result, Statement 7 is returned first to a developer tasked with localizing the statement causing the failing output `mid()`. It is expected that by isolating the statement, the developer will quickly recognize the fault and correct the program. This is the appeal of statistical debugging. It is capable of automatically limiting the number of statements that developers must sort through in the debugging process.

In addition to profiling program statements, most statistical debuggers employ conditional propositions, or *predicates*, to record the values assigned to variables in an execution trace. For example, three predicates for every assignment statement in a program are used to test if a value being assigned to a variable is greater than, less than, or equal to zero. In the context of Statement 7 in the program `mid()`, the three predicates used to test the the value of `m` are $(m > 0)$, $(m < 0)$, and $(m = 0)$.

Although the decision to compare all variable values to zero via these predicates may seem arbitrary, it is effective in general-purpose software. Most general-purpose applications are almost entirely composed of discrete numbers andBoolean conditions. The values that these data types take on are indicators where the sign distinguishes one outcome from another [Liblit 2008].

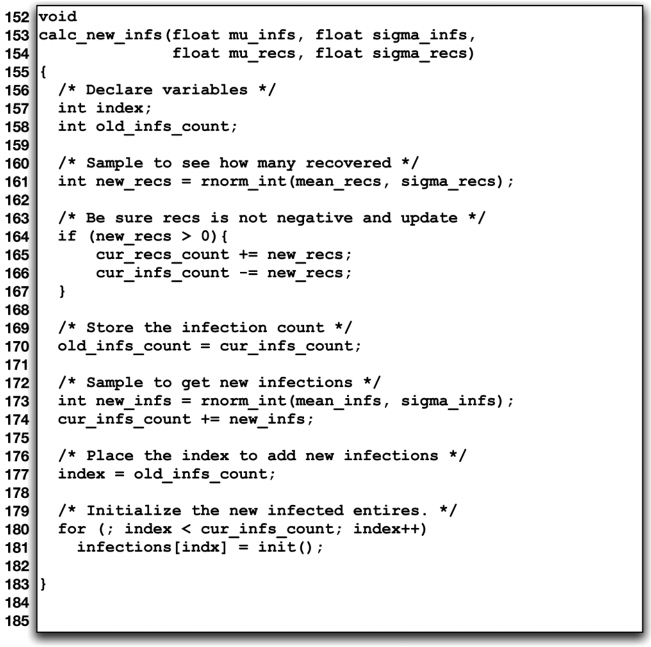
The suspiciousness of these predicates is calculated in the same manner as the suspiciousness of Statement 7. The addition of predicates enables statistical debuggers to analyze the coverage of statements *and* values in execution traces. In theory and practice, this has been shown to improve effectiveness [Liblit 2008].

2.2. Motivating Example

A larger simulation example uncovers the shortcomings of existing statistical debuggers and motivates the improvements that are possible. Figure 2 shows the source code of the `calc_new_infs()` function in the implementation of a published SIR epidemic disease spread model [Gordon 2003]. SIR epidemics divide a population into three states: susceptible (S), infected (I), and recovered (R) [Gordon 2003]. At each timestep in the simulation, the `calc_new_infs()` function is responsible for (1) calculating the number of individuals who were newly infected, (2) calculating the number of individuals who were newly recovered, and (3) initializing a data structure for each newly infected individual.

The logic and use of random variates within the function is an example of a global update in a stochastic, discrete-event simulation [Kelton 2007]. Statement 173 samples a normal distribution centered at `mean_infs` with standard deviation `sigma_infs` to return the number of newly infected individuals. Next, Statement 174 updates the global variable `cur_infs_count`, which tracks the number of currently infected individuals in the population. Finally, Statement 177 places the `index` of the array `infections` into the correct position to start initializing data structures for the newly infected individuals, then statements 180 and 181 perform the initialization.

Here lies the occasionally triggered fault. The size of the `infections` array is set to some large number when the simulation starts up and is not increased. However, because the number of new infections is sampled from a normal distribution, with infinite tails, it is possible to draw an unbounded number of new infections.



```
void
calc_new_infs(float mu_infs, float sigma_infs,
              float mu_recs, float sigma_recs)
{
  /* Declare variables */
  int index;
  int old_infs_count;

  /* Sample to see how many recovered */
  int new_recs = rnorm_int(mean_recs, sigma_recs);

  /* Be sure recs is not negative and update */
  if (new_recs > 0){
      cur_recs_count += new_recs;
      cur_infs_count -= new_recs;
  }

  /* Store the infection count */
  old_infs_count = cur_infs_count;

  /* Sample to get new infections */
  int new_infs = rnorm_int(mean_infs, sigma_infs);
  cur_infs_count += new_infs;

  /* Place the index to add new infections */
  index = old_infs_count;

  /* Initialize the new infected entires. */
  for (; index < cur_infs_count; index++)
    infections[indx] = init();

}
```

Fig. 2. The source code for the `calc_new_infs()` function in a published SIR model [Gordon 2003].

Recognizing that the number of new infections can overrun the statically sized array reveals the fault. When the current number of infections becomes larger than the size of the infections array (due to sampling a distribution with infinite tails), the variable `index` is greater than the room available in `infections` and the program fails. Existing statistical debuggers are unable to effectively identify this fault due to the reliance on static predicates and the assumption that random variates will not be employed. We elaborate on the random variate assumption in Section 2.2.1, then the deficiencies of static predicates are addressed in Section 2.2.2.

2.2.1. Ineffectiveness for Stochastic Simulations. The output of the SIR simulation shown in Figure 2 includes some natural random variance. A single test input can produce slightly different values for the S, I, and R outputs over different executions, and the resulting execution traces are not repeatable. This variance is programmed into the simulation because it is exploratory in nature; the precise output for a given epidemic test case is not exactly known. In contrast, in existing statistical debugging approaches, the labeling of execution traces as passing or failing must be a Boolean function. The disparity between the variance in the SIR simulation and the Boolean labeling functions of existing debuggers creates at least two difficult questions for users tasked with localizing sources of failures in these situations: (1) does an output from an exploratory simulation have to match the analytical solution exactly to be labeled as a passing execution trace? and (2) if not, how close must the output of an execution trace be to the analytical solution to be considered passing and to what extent?

In practice, these questions are ignored entirely, analytical solutions are used as specified outputs, and most execution traces are identified as failing. This profiling process causes most statements to receive the same suspiciousness scores, and the resulting statement rankings incur a significant *Cost* for simulation developers.

Section 3.2 presents an approach inspired by fuzzy logic that enables an execution trace from a simulation employing random variates (i.e., Figure 2) to be labeled as passing and failing. The approach *many-valued labeling functions* removes the requirement that the user must construct a Boolean labeling function. Instead, many-valued labeling functions allow a user to construct a continuous function specifying the extent to which the output of an execution trace passes and fails. Employing many-valued labeling functions enables statistical debuggers to yield statement rankings that incur negligible *Cost* for simulation developers localizing faults within stochastic simulations.

2.2.2. Deficiencies of Static Predicates. Even if the variance from the SIR simulation is removed, existing statistical debuggers still struggle to identify the fault shown in Figure 2. Recall that existing statistical debuggers employ three predicates that partition the values assigned to each variable x in a statement y around zero: $(x_y > 0)$, $(x_y = 0)$, and $(x_y < 0)$. These predicates are referred to as being static, because they are selected before the simulation is executed for any of the supplied test inputs. They assume that the faulty program is largely composed of discrete variables used to encode indicators. Unfortunately, this is not true for most simulations, including the one shown in Figure 2. Almost all of the values assigned to the variables in `calc_new_infs()` are greater than zero and satisfy the same static predicate for each statement, $(x_y > 0)$. As a result, the predicates do not effectively localize the fault.

The elastic predicates presented in Section 3.1 do not assume that variables within a simulation are used as indicators. Instead, elastic predicates adapt to variable values profiled during test execution to cluster together similar values and create predicates that better localize faults in simulations. This approach guarantees that all of the values assigned to a given x_y will not satisfy the same predicate. Many-valued labeling functions and elastic predicates are elaborated on further in Section 3.

3. EXPLORATORY SIMULATION PREDICTION

Two research contributions within our statistical debugger, ESP, enable more effective fault localization for simulations than existing alternatives: elastic predicates and many-valued labeling functions. We elaborate on each contribution in the following. Section 3.1 discusses the philosophy behind elastic predicates and how they are realizable in practice. Then, Section 3.2 describes how fuzzy logic inspires many-valued labeling functions that can categorize a statistical debugging execution trace as both passing and failing and why this enables *Cost*-effective statistical debugging for stochastic simulations.

3.1. Elastic Predicates

Although statistical debuggers employing static predicates have been shown to be effective for a variety of general-purpose software applications, they have done so by ignoring continuous numbers. As a result, they can be improved for simulations that employ continuous numbers or discrete numbers to encode more than signals. Elastic predicates offer such an improvement. Unlike their static counterparts, elastic predicates adapt to profiled variable values to create predicates that better localize faults in simulations.

Given a measure to score suspiciousness, a *maximized elastic predicate* identifies the values assigned to a variable x in a statement y (denoted x_y) that maximize the suspiciousness of the predicate. These maximized elastic predicates are the most effective predicates for fault localization, because each predicate is constructed to identify the most suspicious range of values assigned to each x_y [Liblit 2008].

Unfortunately, generating maximized elastic predicates is impractical. First, they require all of the values assigned to each x_y to be stored and sorted. For many subject programs, the space required to store all of the values assigned to each x_y will exceed the space available on a modern workstation. Furthermore, for most subject programs, sorting the assigned values to each x_y cannot be performed quickly enough to make the resulting maximized elastic predicate useful.

However, other elastic predicates that can be computed efficiently in terms of time and space are realizable. The elastic predicates presented in this section use summary statistics such as the mean, μ_{x_y}, and standard deviation, σ_{x_y}, of the values assigned to a variable x_y to cluster together similar values.

The use of summary statistics to compute elastic predicates is not an arbitrary choice. Summary statistics enable the resulting elastic predicates to be computed in an online manner, which avoids any need to store each value assigned to each x_y [Knuth 1997]. Furthermore, because the summary statistics capture the dispersion of variable values without requiring sorting or optimization routines, the predicates can be computed efficiently. Although elastic predicates based on summary statistics do not maximize the suspiciousness score of each predicate, the results presented in Sections 4 and 5 show they offer significant improvements in effectiveness over the existing alternatives.

Numerous approaches have been proposed to identify predicates that are good failure predictors [Liblit 2008]. The most effective of these approaches analyze variable values within a program. We use two elastic predicate schemes that employ summary statistics such as the mean and standard deviation to analyze variable values. In Section 3.1.1, the elastic *single variable* scheme uses the mean, μ_{x_y}, and standard deviation, σ_{x_y}, of the values assigned to a variable x in a statement y to cluster together similar values. Then, in Section 3.1.2, the *scalar pairs* scheme is presented. The scheme considers the difference between the value of x_y and another in scope similarly typed variable q in statement i to capture the relationships among multiple variables that cannot be detected by the single variable scheme. The elastic scalar pairs scheme uses the mean, $\mu_{x_y-q_i}$, and standard deviation, $\sigma_{x_y-q_i}$, of the difference between the value of x_y and q_i to create partitions that cluster together differences that are similar. Finally, in Section 3.1.3, we return to the motivating example shown in Figure 2 and demonstrate the improvement that elastic predicates offer.

3.1.1. Single Variable. The single variable scheme partitions the set of possible values that can be assigned to a variable x in a statement y. In most debuggers, three static predicates are employed to partition the values for each x_y: $(x_y > 0)$, $(x_y = 0)$, and $(x_y < 0)$. Recall that these three predicates are static because the decision to compare the value of x_y to 0 in each of these predicates is made before the subject program has been executed for any test inputs. Zero is the value chosen for comparison because the sign of a discrete variable is frequently an indicator of the success or failure of an event in general-purpose software applications. Work with automated debugging in the HOLMES project has shown that this is a particularly poor choice for double-precision and floating-point type variables where inevitable numerical analysis errors significantly degrade the value of a comparison to zero [Chilimbi et al. 2009].

In contrast, the nine elastic predicates presented in Table I use summary statistics of the values assigned to x_y during execution to create partitions that cluster together values that are a similar distance and direction from μ_{x_y}. The decision to use nine elastic predicates to partition the values assigned to variable x in statement y for three standard deviations (σ_{x_y}) above and below the mean (μ_{x_y}) is deliberate. This partitioning has been effective to capture the normal dispersion of data for many problems in numerous domains [Bryc 1995]. Furthermore, the partitioning addresses

Table I. Fundamental Single Variable Elastic Predicates

$x_y > (\mu_{x_y} + 3\sigma_{x_y})$
$(\mu_{x_y} + 3\sigma_{x_y}) \geq x_y > (\mu_{x_y} + 2\sigma_{x_y})$
$(\mu_{x_y} + 2\sigma_{x_y}) \geq x_y > (\mu_{x_y} + \sigma_{x_y})$
$(\mu_{x_y} + \sigma_{x_y}) \geq x_y > \mu_{x_y}$
$\mu_{x_y} = x_y$
$(\mu_{x_y} - \sigma_{x_y}) \leq x_y < \mu_{x_y}$
$(\mu_{x_y} - 2\sigma_{x_y}) \leq x_y < (\mu_{x_y} - \sigma_{x_y})$
$(\mu_{x_y} - 3\sigma_{x_y}) \leq x_y < (\mu_{x_y} - 2\sigma_{x_y})$
$x_y < (\mu_{x_y} - 3\sigma_{x_y})$

Table II. Fundamental Scalar Pairs Elastic Predicates

$x_y - q_i > (\mu_{x_y-q_i} + 3\sigma_{x_y-q_i})$
$(\mu_{x_y-q_i} + 3\sigma_{x_y-q_i}) \geq x_y - q_i > (\mu_{x_y-q_i} + 2\sigma_{x_y-q_i})$
$(\mu_{x_y-q_i} + 2\sigma_{x_y}) \geq x_y - q_i > (\mu_{x_y-q_i} + \sigma_{x_y-q_i})$
$(\mu_{x_y-q_i} + \sigma_{x_y-q_i}) \geq x_y - q_i > \mu_{x_y-q_i}$
$\mu_{x_y-q_i} = x_y - q_i$
$(\mu_{x_y-q_i} - \sigma_{x_y-q_i}) \leq x_y - q_i < \mu_{x_y-q_i}$
$(\mu_{x_y-q_i} - 2\sigma_{x_y-q_i}) \leq x_y - q_i < (\mu_{x_y-q_i} - \sigma_{x_y-q_i})$
$(\mu_{x_y-q_i} - 3\sigma_{x_y-q_i}) \leq x_y - q_i < (\mu_{x_y-q_i} - 2\sigma_{x_y-q_i})$
$x_y - q_i < (\mu_{x_y-q_i} - 3\sigma_{x_y-q_i})$

the failings of static predicates for double-precision and floating-point variables where comparison to zero is not useful [Chilimbi et al. 2009].

3.1.2. Scalar Pairs. Multiple variables within a program can have important relationships that cannot be captured with a single variable scheme. Work on the Daikon project has shown that it is useful to identify and capture the relationships among multiple variables with simple and implicit invariants that aid program evolution and program understanding [Ernst et al. 2001]. Similarly, statistical debuggers capture important relationships among multiple variables by identifying invariants that are only violated when the subject program fails. The scheme that captures these invariants is the scalar pairs scheme [Liblit 2008].

In the static scalar pairs scheme, at each assignment to a variable x in a statement y, all other in-scope, same-typed local or global variables $q_1, q_2, \ldots, q_i, \ldots, q_n$ are identified. For each pair of variables, the static scalar pairs scheme compares the difference of a new value for x_y and the existing value of q_i to zero: $(x - q_i > 0)$, $(x_y - q_i = 0)$, $(x_y - q_i < 0)$.

In contrast, the nine scalar pair elastic predicates presented in Table II use summary statistics of the difference between the new value of x_y and the existing value of q_i to create partitions that cluster together differences that are a similar distance and direction from $\mu_{x_y-q_i}$.

3.1.3. Revisiting the Motivating Example with Elastic Predicates. The utility of a statistical debugger is determined through experimental evaluation. However, one can begin to appreciate the improvements offered from elastic predicates by applying them to the motivating example that we presented in Section 2.2.

Recall that even if we fixed the seed of each random variate in the SIR simulation shown in Figure 2, the fault is still difficult to localize with static single variable predicates. The difficulty is created because almost all of the values assigned to the variables in the function satisfy the predicate $(x_y > 0)$. However, the single variable

Table III. The Top-Ranked Predicates for Figure 2

Filename	Elastic/Static	Function	Predicate
`sir.c`	Elastic	`calc_new_infs()`	$index_{180} > (\mu_{index_{180}} + 3\sigma_{index_{180}})$
`sir.c`	Static	`calc_new_infs()`	$cur\_infs\_count_{174} > 0$

elastic predicates in Table I avoid this problem. Profiling the values assigned to each x_y during execution creates partitions that ensure only similar variable values, in terms of distance and direction from the mean, satisfy the same predicate.

The most suspicious static predicate and the most suspicious elastic predicate for the `calc_new_infs()` function are shown in Table III. The predicates and their suspiciousness estimates are computed using 4,000 randomly generated, valid test inputs to the SIR simulation. Table III shows that the elastic predicate ($index_{180} > (\mu_{index_{180}} + 3\sigma_{index_{180}})$) clusters together unusually large values assigned to the variable `index` in Statement 180 and captures the fault. The predicate suggests that failures frequently occur when the parameters supplied to the `calc_new_infs()` function generate an unusually large number of infected individuals. Specifically, failures occur when the `infections` array does not have room for the number of generated infected individuals and their corresponding data structures. The location of the fault and the cause of the failure are clear after identification and explanation. However, this fault was present and undiscovered in the simulation for several years [Gore 2012].

In contrast to the top-ranked elastic predicate, no single variable static predicates are particularly suspicious. The most suspicious single variable static predicate suggests that some values assigned to the variable `cur_infs_count` in statement 174, which are greater than zero, cause the program to fail. Although the suggestion is correct, it does not lead the developer to the location or the direct cause of the fault. The top-ranked elastic predicate does.

This example showcases the opportunity to improve statistical debuggers by augmenting them with elastic predicates. However, combining the elastic predicates with the many-valued labeling functions described in Section 3.2 enables further improvements to an even larger class of simulations.

3.2. Many-Valued Labeling Functions

Simulations are frequently employed to explore domains where physical experimentation is impossible due to economic, moral, and ethical constraints. In these cases, the expected output for a given test input is not known and thus impossible to provide. Even if analytical solutions are used to provide the expected output, some simulations include random variates and thus variance in their output.

For example, we generated 4,000 random valid test inputs to the SIR simulation shown in Figure 2, and despite many execution traces approaching their analytical solution, none of the traces matched it. As a result, each of the execution traces was labeled as failing, all of the program statements were given the same suspiciousness score, and the *Cost* required to debug the simulation shown in Figure 2 was overwhelming. This example reveals the debilitating nature of the statistical debugger requirement that a set of test inputs, corresponding execution traces, and a labeling of the execution traces as passing or failing be provided by the user.

Here we present many-valued labeling functions that provide a framework to relax this requirement. Many-valued labeling functions enable each SIR execution trace to both pass and fail to a given extent. This removes the requirement that the user must construct a Boolean function to label the output of an execution trace as either strictly passing or failing. As a result, many-valued labeling functions enable different program statements to receive different suspiciousness scores generating a ranked list of program statements that incur significantly less *Cost* for simulation developers.

3.2.1. Formal Definition. Our many-valued labeling functions assume that the output of a program is (or can easily be transformed to) some ordered, separate, set of real numbers. This is typical of most simulations. This set is referred to as the output list, X. The passing extent, u, of the output list, X, is computed using Equation (1):

$$u = \left(\frac{\sum_{i=1}^{|X|} W_i f_i(X_i)}{\sum_{i=1}^{|W|} W_i} \right) \text{where } f_i : \mathbb{R} \to [0,1] \text{ and } |W| = |X|. \tag{1}$$

In Equation (1), W is an ordered list of weights, which focuses attention on particular parts of the output list. The functions f_i encode information about the specified (or passing) output and the tolerance for deviation from that output. Note that $f(X)$ is separable into $f_i(X_i)$ due to our assumption that the output of a program is an ordered, separated, set of real numbers. The passing extent u reflects the extent to which executing and tracing the simulation for a single test input produces the specified output. Similarly, the complement of the passing extent, the failing extent $(1.0 - u)$, reflects the extent to which executing and tracing the simulation for a single test input does not produce the specified output.

The sum of the failing extent of every execution trace that includes a given predicate or statement in the test suite reflects the total number of failing execution traces for that predicate. This sum and the total number of execution traces including the statement or predicate are used to compute suspiciousness. Recall that suspiciousness is calculated by computing the ratio of the number of failing execution traces that include the statement (or predicate) to the number of total execution traces that include the statement (or predicate).

3.2.2. Backward Compatibility. Many-valued labeling functions are a continuous generalization of Boolean labeling functions. This means that although they do not guarantee improvements in the effectiveness of statistical debuggers for stochastic simulations, they can reproduce the results of Boolean labeling functions. This backward compatibility is described next.

Consider a simulation where the output for each execution trace is mapped to a single real number X, and the passing output for the execution trace is the real number X_{pass}. The delta function function, $f(x) = \delta_{X_{pass}}(X)$, will separate execution traces that pass $(u = 1)$ from execution traces that fail $(u = 0)$. The choice of $f(x)$ is important: $\delta_{X_{pass}}(X) = 1$ if X exactly matches X_{pass}; otherwise, $\delta_{X_{pass}}(X) = 0$.

3.2.3. Choosing the Appropriate Many-Valued Function. Ultimately, choosing the functions for the passing/failing extent for the execution traces of a faulty simulation is a nontrivial problem. Currently, it entirely depends on the simulation developer. The examples discussed in Section 4.1.1 and our case study in Section 6 serve as possible templates for users to pursue, but we cannot provide a one-size-fits-all solution. However, the evaluation in Section 4 shows that the many-valued formalism *allows* a choice that can outperform the Boolean function used to label passing and failing execution traces in existing statistical debugging approaches. Furthermore, although many-valued labeling functions do not guarantee improvements, Section 3.2.2 shows that they can reproduce the results of the existing Boolean functions used to label passing and failing execution traces. This ensures that ESP, which employs many-valued labeling

Table IV. Subject Programs Used in the Evaluation of Elastic Predicates

Name	Number of Versions Used / Number of Versions	LoC	Number of Tests	Description
`schedule`	9/9	292	2,710	Priority scheduler
`scheule2`	9/10	301	2,650	Priority scheduler
`tcas`	41/41	141	1,608	Altitude separator
`bates`	1/1	8,184	298	Options pricing model
`heston`	1/1	4,095	316	Options pricing model
`mc euro`	1/1	835	242	Options pricing model
`um-olsr`	1/1	14,433	176	Network protocol simulator
`ns2`	1/1	11,258	293	Network simulator
`g/g/1`	1/1	2,247	3,000	Queuing simulation
`m/m/c`	1/1	3,302	3,000	Queueing simulation
`mmpp/d/1`	1/1	3,860	3,000	Queueing simulation
`ising`	1/1	875	3,000	Physics simulation
`sir`	1/1	5,892	4,000	Disease spread simulation

functions reasonably, will not be outperformed by existing statistical debuggers, which do not.

4. EVALUATION

The utility of a fault localization approach is determined through experimental evaluation. In this section, a thorough evaluation of the effectiveness and the efficiency of elastic predicates and many-valued labeling functions is presented. Sections 4.1 and 4.2 describe the subject simulations and fault localization approaches included in the evaluation. The effectiveness of the approaches is evaluated in Section 4.3. Then, the efficiency of all approaches is presented in Section 4.4.

4.1. Subject Simulations

We include 13 subject simulations in our evaluation, yielding a total of 69 different faulty versions. The simulations include three adapted fault localization benchmarks (`schedule`, `schedule2`, `tcas`) and 10 widely used simulations with actual faults observed in the wild (`sir`, `ising`, `bates`, `heston`, `mc euro`, `um-olsr`, `ns2`, `g/g/1`, `m/m/c` and `mmpp/d/1`).

Table IV shows the characteristics of the subjects. For each subject, the first column gives the simulation name, the second column provides the ratio of the number of versions used to the number of versions available, the third column gives the number of lines of code for the subject, the fourth column gives the number of tests, and the last column provides a description. One of the faulty versions of the `schedule2` simulation did not contain syntactic differences between the correct version and the faulty version. As a result, this version was omitted from the evaluation.

4.1.1. Adapted Benchmarks. The three established benchmarks included in the evaluation suite are common objects of analysis in the statistical debugging community. Each benchmark contains a number of faulty versions with a set of test inputs and expected outputs for each version. In this evaluation, we adapted the benchmarks to include random variates. For the `tcas` program, the values of constants are sampled from a uniform distribution where the minimum value is half of the value of the constant and the maximum value is one and a half the value of the constant.

In the two priority schedulers (`schedule` and `schedule2`), the programs are modified to include arrival and service times drawn from a normal distribution for each process. The unmodified test inputs for these programs dictate the order in which processes are

Table V. The Many-Valued Labeling Function Used for `tcas` Test Cases

Expected Output	Faulty Version Output	Passing Extent (u)	Failing Extent ($1.0 - u$)
0	0	1.0	0.0
0	1	0.5	0.5
0	2	0.0	0.0
1	0	0.5	0.5
1	1	1.0	0.0
1	2	0.5	0.5
2	0	0.0	1.0
2	1	0.5	0.5
2	2	1.0	0.0

scheduled in the queue. Our modifications represent the uncertainty of measurements often used in simulations that create variance in program output. However, each faulty version of each program contains the same faults and same inputs as the original faulty version in the benchmark. These programs were chosen because straightforward adaptations could be made to create simulations that featured (1) some variance in the output and (2) faults from an established set of benchmarks.

The Traffic Collision Avoidance System, `tcas`, monitors an aircraft's airspace and warns pilots of possible collisions via three different outputs: (0) adjust the trajectory of the aircraft downward, (1) do not adjust the trajectory of the aircraft, or (2) adjust the trajectory of the aircraft upward. For the approaches employing Boolean labeling functions, a test case for a faulty version of `tcas` passes if it directly matches the output that was specified for the deterministic version of the `tcas` program. For the approaches employing many-valued labeling functions, the function, f, shown in Table V is employed. The function shown in Table V does not consider an execution trace to fully fail unless it drastically differs from the specified output. This occurs when a faulty version instructs the pilot to (a) adjust the trajectory of the aircraft downward when an upward adjustment is specified or (b) adjust the trajectory of the aircraft upward when a downward adjustment is specified.

Although Table V shows a many-valued labeling function for three outcomes, its structure can be extended to any simulation with a finite number of expected outcomes. Example simulations with an infinite number of expected outcomes are discussed in Section 4.1.2 and in our case study in Section 6.

The output of the two scheduling benchmarks (`schedule` and `schedule2`) is a list of the identification numbers of processes in the order they exit the system. For the approaches employing Boolean labeling functions, a test case passes if the ordered output of the faulty version exactly matches the ordered output specified for the test case. If this is not true, then the test case fails. However, for the approaches employing many-valued labeling functions, the Levenshtein distance between the ordered output of the faulty version and the ordered output specified for the test case serves as the labeling function. The Levenshtein distance between the output and the specified output is the minimum number of edits needed to transform one list of process identification numbers into the other, where the edit operations are insertion, deletion, or substitution of process identification numbers [Navarro 2001].

4.1.2. Widely Used Simulations. Each of the widely used simulations contains one fault that reflects a documented error that has been observed by subject matter experts (SMEs). For example, the Bates stochastic volatility jump-diffusion pricing simulation (`bates`) must be calibrated to previous data before it is employed to make price predictions for the future. However, if the absolute price error is minimized during calibration instead of the relative price error, the simulation produces an error [Detlefsen and Hardle 2007]. Similarly, the implementation of the Heston stochastic volatility simulation (`heston`) reflects documented issues in the computation of the logarithms for complex numbers [Mikhailov and Nögel 2003]. The pricing simulation of European barrier options (`mc euro`) contains an error in computation of bank offering rates [Boyle and Lau 1994]. The `um-olsr` protocol used with the `ns2` network simulator contains a documented (and now patched) error in the degree method [Ros 2007]. In the 2.19b version of the `ns2` network simulator, which can be used to simulate bandwidth usage for implementations of the TCP protocol, there is a fault that incorrectly tracks the number of nodes in the network [Issariyakul and Hossain 2011].

The three queueing simulations included in the evaluation each contain published faults related to the misuse of uncertainty [Kelton 2007] (each of these simulations is described further in Gross et al. [2011]). 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. The second is a `m/m/c` queueing simulation with a misused Poisson distribution. The third is a `mmpp/d/1` queueing simulation with an incorrectly bound loop due to the misuse of a random variate.

The last two simulations included in the evaluation are `sir` and `ising`. The `sir` simulation and its fault are discussed in depth in Section 2.2. Finally, the `ising` simulation is a Monte Carlo implementation of the canonical Ising model used in physics [Brush 1967]. However, it does not take the absolute value of the magnetization of each spin creating a fault.

Established analytical solutions for each of these simulations exist in published research. However, recall even if analytical solutions are used to provide the expected output, random variates in simulations create variance in their output, rendering traditional statistical debuggers largely ineffective. Here we use these analytical solutions to parameterize a many-valued passing function, f, for all of the simulations. The function f is a bell curve centered on the analytical solution, $\bar{x}$, for the test inputs that accompany each simulation. Formally, f is $\exp(-(x-\bar{x})^2/a^2)$, where a is the tolerance for deviation. We use $a = 1.0$, and the extent to which each execution trace matches its specified output of f is reflected by the variable, u. Similarly, the extent to which each execution trace fails is $1.0 - u$.

4.1.3. The Nature of the Faults. All of the faults included in the subject simulations are computation related as opposed to memory related. These faults reflect operator and operand mutations, missing and extraneous code, and constant value mutations. For simulation versions with a faulty constant assignment statement, the assignment statement is considered to be examined by a developer when it is directly examined or when a statement explicitly using the constant is examined. Second, for simulation versions where the fault reflects a missing statement, statements directly adjacent to the missing code qualify as the missing statement. These issues are handled the same way in other published research in the statistical debugging community [Jones and Harrold 2005; Jeffrey et al. 2008].

4.2. Competing Approaches

Three different general approaches to fault localization are featured in the evaluation: Cooperative Bug Isolation (CBI), ESP, and Tarantula. However, for each general approach, a version that employs (1) Boolean labeling functions and (2) many-valued labeling functions is included. The versions employing many-valued labeling functions are referred to as being fuzzy to distinguish them from their traditional counterparts. Here we summarize the similarities and the differences of the strategies employed in the six approaches included in our evaluation.

4.2.1. Cooperative Bug Isolation. In the evaluation, CBI is employed with the static single variable and static scalar pairs predicates described in Sections 3.1.1 and 3.1.2. The suspiciousness of a predicate is calculated by computing the ratio of failing execution traces that cover a predicate to total execution traces that cover the predicate. Given a list of CBI predicates ranked by suspiciousness, program statements are ranked according to the following:

(1) For each statement, *stmt*, identify the corresponding predicate with the highest suspiciousness score, $stmt_{high}$.
(2) Move the statement, *stmt*, and highest suspiciousness score, $stmt_{high}$, to set ST.
(3) Rank the statements in ST in descending order by suspiciousness score.

4.2.2. Fuzzy CBI. In the traditional version of CBI, a Boolean decision is made as to whether or not an execution trace is labeled as passing or failing. This decision entirely hinges on whether the output of an execution trace exactly matches what is specified for the test input.

However, in the fuzzy version of CBI, the extent, u, to which an execution trace is labeled passing is determined by a many-valued function. Accordingly, predicate suspiciousness is determined by (1) summing the failing extent $(1.0 - u)$ for every execution trace that covers the predicate and (2) computing the ratio of that sum and the total number of execution traces that cover the predicate. Fuzzy CBI and CBI employ the same predicates and rank them in the same manner.

4.2.3. Exploratory Simulation Prediction. The predicates employed in ESP are a superset of those employed in CBI. Along with the static single variable and static scalar pairs predicates employed in CBI, ESP also employs the elastic single variable and the elastic scalar pairs predicates described in Sections 3.1.1 and 3.1.2. The suspiciousness and ranking of these predicates is computed in the same manner as in CBI.

4.2.4. Fuzzy ESP. The predicates employed in fuzzy ESP are they same as those employed in ESP. The labeling of execution traces, the suspiciousness of predicates, and the ranking of the program statements in fuzzy ESP are computed in the same manner as they are in Fuzzy CBI.

4.2.5. Tarantula. In contrast to the predicative-level statistical debuggers CBI and ESP, Tarantula is a statement-level statistical debugger. A detailed description of how Tarantula works is presented in Figure 1. For a given statement, Tarantula profiles the number of failing execution traces that cover the statement and the total number of execution traces that cover the statement. The suspiciousness score of the statement is calculated by computing the ratio of failing execution traces to the total execution traces. Execution traces are labeled as passing or failing using Boolean functions in Tarantula, just as they are in the traditional versions of CBI and ESP. Program statements are ranked in descending order of suspiciousness [Jones and Harrold 2005].

4.2.6. Fuzzy Tarantula. In this version of Tarantula, many-valued passing functions are used to determine the extent, u, to which an execution trace passes or fails for a given test input. Statement suspiciousness is determined by (1) summing the failing extent $(1.0 - u)$ of every execution trace that covers the statement and (2) computing the ratioof the sum and the total number of execution traces where the statement is covered. Statements are ranked in the same manner as Tarantula.

4.3. Effectiveness

To study the effectiveness of employing elastic predicates and many-valued labeling functions in the approaches described in Section 4.2, a version of the established metric *Cost* is employed [Jones and Harrold 2005; Jeffrey et al. 2008]. Given a ranked set of statements, *Cost* measures the percentage of executed statements that a developer must examine before encountering the faulty statement. If there are ties, it is assumed that the developer must examine all of the tied statements. For example, if there are n

Table VI. Number (Percentage) of Faulty Version Ranked Statement Lists in Each Score Range for All Approaches

$\overline{Cost}$	Tarantula # (%) Sims	Fuzzy Tarantula # (%) Sims	CBI # (%) Sims	Fuzzy CBI # (%) Sims	ESP # (%) Sims	Fuzzy ESP # (%) Sims
0%–1%	1 (1.44%)	3 (4.34%)	3 (4.34%)	7 (10.14%)	4 (5.80%)	9 (13.04%)
1%–10%	5 (7.25%)	15 (21.73%)	7 (10.14%)	22 (31.88%)	9 (13.04%)	26 (37.68%)
10%–20%	7 (10.14%)	12 (17.39%)	10 (14.44%)	10 (14.44%)	11 (15.94%)	10 (14.44%)
20%–30%	7 (10.14%)	6 (8.69%)	8 (11.59%)	6 (8.69%)	8 (11.59%)	5 (7.25%)
30%–40%	11 (15.94%)	7 (10.14%)	9 (13.04%)	4 (5.80%)	9 (13.04%)	4 (5.80%)
40%–50%	7 (10.14%)	7 (10.14%)	7 (10.14%)	5 (7.25%)	9 (13.04%)	3 (4.34%)
50%–60%	8 (11.59%)	5 (7.25%)	7 (10.14%)	3 (4.34%)	7 (10.14%)	3 (4.34%)
60%–70%	9 (13.04%)	5 (7.25%)	8 (11.59%)	4 (5.80%)	4 (5.80%)	2 (2.89%)
70%–80%	6 (8.69%)	3 (4.34%)	4 (5.80%)	3 (4.34%)	3 (4.34%)	3 (4.34%)
80%–90%	4 (5.80%)	3 (4.34%)	3 (4.34%)	3 (4.34%)	3 (4.34%)	2 (2.89%)
90%–100%	4 (5.80%)	3 (4.34%)	3 (4.34%)	2 (2.89%)	2 (2.89%)	2 (2.89%)

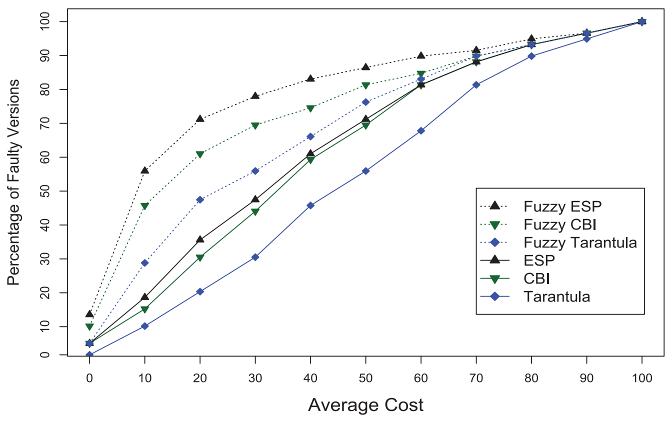


Fig. 3. Evaluation of the effectiveness of many-valued labeling functions and elastic predicates. Higher and farther to the left is better.

executed statements in a program and all n statements have the same suspiciousness score, it is assumed that the developer must examine all n statements. A lower score is preferable, because it means that fewer statements must be considered before the faulty statement is found. Due to the variance in the output of the simulations used in the evaluation, a version of *Cost* that reflects the average *Cost* of localizing the fault in the simulation over 100 different trials is used. This cost metric is referred to as $\overline{Cost}$.

The effectiveness of employing elastic predicates and many-valued labeling functions in the approaches included in the evaluation for the subject simulations is shown in Table VI. Each row in Table VI shows a $\overline{Cost}$ range and the number (and percentage) of subjects for each approach that incur the specified $\overline{Cost}$. Figure 3 provides a graphical view of this data. In this figure, the x-axis represents the lower bound of each $\overline{Cost}$ range, and the y-axis represents the percentage of subjects where a $\overline{Cost}$ less than or equal to the upper bound is incurred. This presentation of data follows the established convention in the statistical debugging community [Jones and Harrold 2005; Jeffrey et al. 2008].

4.3.1. CBI and Fuzzy CBI. Fuzzy CBI performs better than CBI for the simulations included in the evaluation. There are only 2 out of the total 69 faulty simulation versions where Fuzzy CBI assigned a lower rank to the statement containing the fault than CBI. In these versions, the many-valued labeling functions used to calculate the suspiciousness scores do not localize the faulty statement as well as their traditional counterparts. These are outlying data points in the evaluation ($<3\%$ of all faulty versions). For most of the faulty versions, the many-valued labeling functions used to label execution traces as passing or failing in Fuzzy CBI lead to more effective fault localization. Furthermore, for the two versions where CBI outperforms Fuzzy CBI, the faults in the versions reflect missing code. Other published attempts to improve the effectiveness of statistical debuggers have failed to localize these types of faults well [Baah et al. 2011].

Overall, the traditional version of CBI struggles in this portion of the evaluation. It employs static predicates, and as a result many values assigned within the simulations satisfy the same predicates. This hurts the approach. ESP does not fall victim to the same problem because of the elastic predicates that it employs. Furthermore, the variability in the output of the simulations results in CBI classifying significantly more execution traces as failing than does Fuzzy CBI. The combination of static predicates and a plethora of failing execution traces results in all of the predicates being assigned the same or very similar suspiciousness scores. These similarities and ties in suspiciousness result in poor $\overline{Cost}$ scores because very few predicates are separated from the masses.

4.3.2. ESP and Fuzzy ESP. The results of the evaluation for ESP and Fuzzy ESP are encouraging. They enable ESP to outperform CBI and Fuzzy ESP to significantly outperform Fuzzy CBI. However, the relative difference between ESP's performance against CBI and Fuzzy ESP's performance against Fuzzy CBI deserves further investigation. The traditional version of ESP classifies significantly more test cases as failing than does Fuzzy ESP. As a result, some of the elastic predicates in the traditional version of ESP are assigned the same or similar suspiciousness score, just as were the static predicates in CBI. Ultimately, the predicates with the same suspiciousness scores render traditional ESP less effective than Fuzzy ESP. However, it is still important to note than when comparing the nonfuzzy implementations, ESP outperforms CBI by a noticeable margin, and when comparing Fuzzy ESP to Fuzzy CBI the margin of victory is significant.

4.3.3. Fuzzy Tarantula. Fuzzy Tarantula also outperforms its traditional counterpart in the evaluation. Recall that both versions of Tarantula only profile statement coverage data as opposed to the traditional and fuzzy versions of ESP and CBI, which employ predicates to profile variable values. As a result, the traditional version of Tarantula is less effective than the traditional version of ESP or CBI. In addition, the fuzzy version of Tarantula is less effective than the fuzzy version of ESP or CBI.

It is important to note the particularly poor performance of the traditional Tarantula approach in this portion of the evaluation. The performance of Tarantula is worse than the expected performance of a simple-minded approach to fault localization, which takes all statements included in failing execution traces and ranks them in a random order. Such an approach would be expected to localize faults in the same percentage of faulty versions as the $\overline{Cost}$ incurred. For example at a $\overline{Cost}$ of searching through 50% of the source code in each of the faulty versions, the simple-minded approach would be expected to localize faults in 50% of the faulty versions. Similarly, at a $\overline{Cost}$ of searching through 30% of the source code of the faulty versions, it would be expected to localize faults in 30% of the faulty versions.

Table VII. Average Wallclock Time (in Seconds) Required by Each Approach to Execute All Faulty Versions of the Specified Subject Program over 100 Trials

Name	CBI (Traditional/Fuzzy)	ESP (Traditional/Fuzzy)	Tarantula (Traditional/Fuzzy)
`schedule`	757/707 ∼ (1.07)	2,088/2,039 ∼ (1.02)	81/30 ∼ (2.7)
`scheule2`	658/610 ∼ (1.07)	2,077/2,030 ∼ (1.02)	79/30 ∼ (2.63)
`tcas`	186/183 ∼ (1.01)	864/861/864 ∼ (1.00)	15/12 ∼ (1.25)
`bates`	579/564 ∼ (1.02)	2,331/2,293 ∼ (1.01)	49/35 ∼ (1.4)
`heston`	984/961 ∼ (1.02)	3,984/3,941 ∼ (1.01)	68/56 ∼ (1.21)
`mc euro`	648/626 ∼ (1.03)	2,681/2,624 ∼ (1.01)	49/40 ∼ (1.22)
`um-olsr`	1,746/1,728 ∼ (1.01)	1,825/1,802 ∼ (1.01)	41/30 ∼ (1.37)
`ns2`	1,327/1,285∼ (1.03)	5,891/5,857 ∼ (1.00)	92/80 ∼ (1.15)
`g/g/1`	369/353 ∼ (1.03)	1,562/1,545 ∼ (1.01)	34/23 ∼ (1.48)
`m/m/c`	319/307 ∼ (1.04)	1,346/1,328 ∼ (1.01)	26/18 ∼ (1.44)
`mmpp/d/1`	462/449 ∼ (1.02)	2,322/2,284 ∼ (1.01)	42/29 ∼ (1.44)
`sir`	626/604 ∼ (1.04)	2,602/2,561 ∼ (1.01)	53/38 ∼ (1.39)
`ising`	346/328 ∼ (1.05)	1,303/1,285 ∼ (1.01)	27/21 ∼ (1.29)

The traditional version of Tarantula does not outperform the simple-minded approach because there are no ties in terms of suspiciousness estimates in the ranked list of statements in the simple-minded approach. In the traditional version of Tarantula, ties occur frequently in a ranked list of statements in this portion of the evaluation because many execution traces are classified as failing. Since the $\overline{Cost}$ metric requires all tied statements to be examined before the fault is considered to be localized, the effectiveness of Tarantula in this portion of the evaluation is particularly poor.

4.3.4. Overall Effectiveness. The demonstrable improvement in performance of the fuzzy version of each traditional debugger is significant. Without many-valued labeling functions, statistical debugging incurs so much $\overline{Cost}$ that it is inapplicable to simulations where a Boolean determination of passing and failing cannot be made. However, by enabling users to specify many-valued labeling functions for these simulations, the effectiveness of the fuzzy debuggers approaches that of their traditional counterparts when applied to general-purpose software. Ultimately, many-valued labeling functions enable statistical debugging to be applicable to an entire set of simulations where previously users had no alternative for automatic fault localization.

4.4. Efficiency

Here, the efficiency of the elastic predicates and many-valued labeling functions employed in the evaluation is analyzed. Table VII shows the average wallclock time for ranking the statements for each faulty version of the subjects. The *Slowdown* incurred by employing many-valued labeling functions for each general approach is shown in parentheses. *Slowdown* is calculated by computing the ratio of the *fuzzy* runtime to the *traditional* runtime.

4.4.1. Elastic Predicates. The generation and analysis of elastic predicates in ESP (fuzzy and traditional) does not create constant overhead in terms of execution time, but the additional time required to employ the elastic predicates is never greater than five times the execution time of static predicate employing CBI (fuzzy and traditional). The scalar pairs predicates, as opposed to the single variable predicates, in CBI and ESP require most of the execution time. Employing static scalar pairs predicates in CBI can result in an execution time that is more than 15 times slower than applying Tarantula (fuzzy and traditional) to the same subject program. Furthermore, for some of the simulations included in the evaluation, employing both static and elastic scalar pairs predicates in ESP results in an execution time that is more than 50 times greater than that of Tarantula. However, these efficiency results (~5 times slower than CBI, ~50 times slower than Tarantula) seem reasonable considering the ability of elastic predicates to provide simulation developers with a significantly better fault localization approach.

4.4.2. Many-Valued Labeling Functions. The most evident trend in Table VII related to the use of many-valued labeling functions is that the *Slowdown* incurred very much depends on the function chosen by the user. The simpler labeling functions used by the approaches for localizing faults in the `tcas`, `bates`, `heston`, `mc euro`, `um-oslr`, `ns2`, `g/g/1`, `m/m/c`, `mmpp/d/1`, `sir`, and `ising` simulations incur less *Slowdown* than the complex Levenshtein distance function employed for `schedule` and `schedule2`.

The increase in *Slowdown* when moving from the simple functions to the more complex Levenshtein distance is more evident in Fuzzy Tarantula than in Fuzzy CBI or Fuzzy ESP. This is because in Fuzzy ESP and Fuzzy CBI, the additional computational overhead incurred for the labeling function is masked by the profiling used to collect predicate data and score suspiciousness. The statement-level implementation of Fuzzy Tarantula does not employ predicates and thus does not mask the inefficiencies caused by complex labeling functions.

4.4.3. Machine Time Versus Developer Time. It is important to note that although elastic predicates and many-valued labeling functions make Fuzzy ESP less efficient to run than Fuzzy CBI and Fuzzy Tarantula, these research contributions only incur machine time and not developer time. If developers can remain productive while Fuzzy ESP generates its ranked list of predicates, overall efficiency will be significantly improved because the developer is given a drastically more effective list of ranked statements to debug the simulation.

However, for simulations where the runtime is so overwhelming that increasing it by a factor of 50 or 100 times would create unproductive developers, then using ESP would be impractical. In general, we believe that these simulations are rare. Multiplying the execution time of most software by a scalar factor rarely changes overall business practices [Brat et al. 2000].

5. ESP IN AN IMPERFECT WORLD

The effectiveness of elastic predicates and many-valued labeling functions enable ESP to incur significantly less *Cost* than CBI or Tarantula. However, there are three factors that were controlled in our previous evaluation and could contribute to ESP's effectiveness: (1) the rate at which simulation predicates are sampled, (2) the number of test inputs supplied for each simulation, and (3) the number of faults present in each simulation.

In Section 4, we conducted an evaluation where every predicate instrumented in a simulation was *always* sampled, *thousands* of test inputs were available for each simulation, and in most simulations *only one* fault was present. Although this environment is highly desirable, it is not always available in practice. Therefore, we consider our Section 4 evaluation environment a *perfect world*. In Sections 5.1 through 5.3, we explore (1) the extent to which ESP benefited from the perfect world conditions and (2) the robustness of ESP when the conditions degrade in an imperfect world.

It is important to note that when ESP and CBI are referred to throughout this section, the fuzzy versions of the respective approaches are invoked. Given the superior effectiveness of the fuzzy approaches *and* the backward compatibility to their traditional counterparts, it is unnecessary to explore the traditional versions further for the simulations employed in our evaluation.

5.1. Sparse Sampling

Recall that ESP and CBI use source code instrumentation to profile predicates in programs and simulations. The instrumentation, profiling, and subsequent analysis adds

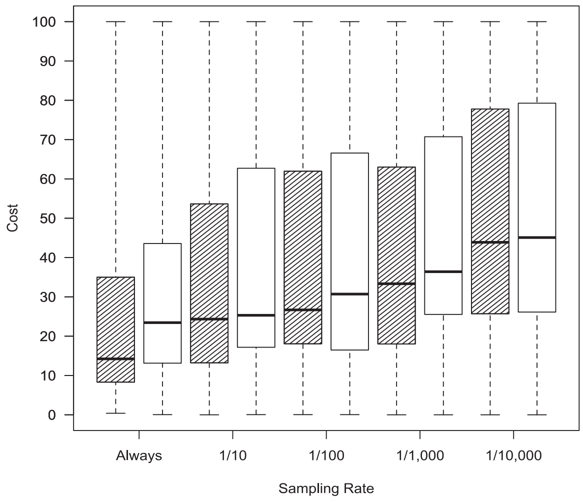


Fig. 4. CBI (nonstriped) and ESP (striped) under sparse sampling rates for the subject programs.

overhead to the execution. In CBI, this overhead can be limited by employing random sampling of the predicates rather than always profiling every predicate. The sampling is unbiased, collecting a representative subset of all predicates across the subject program test suite. To ensure sufficient data collection, CBI relies on the large user communities of the software in which it is deployed. The result is an effective approach to isolating faults in software with wide distribution [Liblit 2008].

ESP is not designed to meet the same goals. In the most common use case, ESP is deployed as a stand-alone approach for a single simulation developer. In this use case, the goal is to identify failure-predicting predicates as effectively as possible for the test inputs provided. As a result, random sampling is not used in ESP. However, to explore the environments in which ESP is useful, it is important to evaluate elastic predicates and many-valued labeling functions in the context of random sampling.

The $\overline{Costs}$ of 100 executions of the subject programs included in the evaluation under ESP (striped) and CBI (nonstriped) with sampling rates from 1/10 to 1/10,000 are plotted in Figure 4. The bottom and top of each box in Figure 4 represent the lower and upper quartile $\overline{Cost}$ and the black band is the median $\overline{Cost}$. The whiskers extend to the lowest and highest $\overline{Cost}$.

Figure 4 shows that the effectiveness of CBI remains more stable under sampling rates of 1/10 and 1/100 than the effectiveness of ESP. Once sampling is introduced, the median $\overline{Cost}$ of localizing a fault in ESP increases significantly. Although ESP continues to remain more effective than CBI by an absolute margin, the relative difference in effectiveness between the two approaches narrows. At a sampling rate of 1/1,000, both ESP and CBI begin to become significantly less effective.

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 profiled variable is based on so little data that the resulting elastic predicates are no better, and often worse, than the static predicates employed in CBI 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 such as 1/100, estimations of the mean and standard deviation result in effective failure-predicting predicates. This is significant: sampling rates of

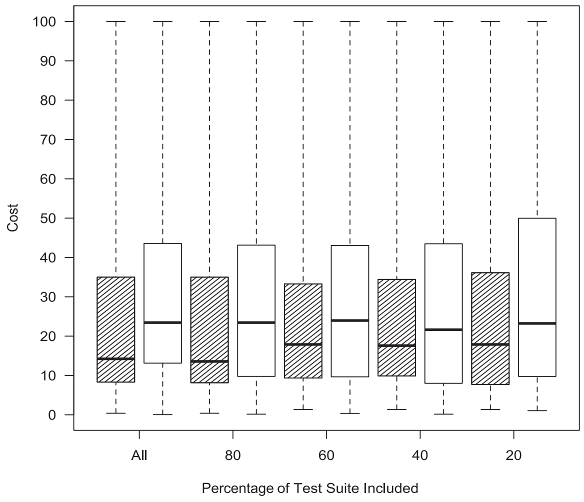


Fig. 5. CBI (nonstriped) and ESP (striped) with incomplete test suites for the subject programs.

approximately 1/10 do not necessarily reduce overhead, whereas rates of 1/100 or more do [Liblit 2008].

Given these results, it would be practical to consider applying sampling to reduce the overhead of ESP for large distributed simulations with multiple users. In the worse case, the effectiveness of ESP would reduce to CBI, and for sampling rates of 1/1,000 or more, we would expect ESP to outperform CBI.

5.2. Incomplete Test Suites

Uncertainty in data collection can also be introduced through an incomplete or sparse test suite. Each subject included in our evaluation is accompanied by several thousand test inputs. Although this is desirable, it rarely occurs in practice because of how simulations are developed and tested [Law and Kelton 1991; Krahl 2005]. Figure 5 summarizes the effect of smaller test suites on the elastic predicates and many-valued labeling functions in ESP (striped) compared to the static predicates and many-valued labeling functions in CBI (nonstriped). At random, 100 incomplete test suites from the original test suite for each subject program were chosen, forming test suites at $1/5^{\text{th}}$, $2/5^{\text{th}}$, $3/5^{\text{th}}$, and $4/5^{\text{th}}$ 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 10 failing test cases and at least 20 successful (passing) test cases, it was dissolved and the sparse test suite was reformed. The effectiveness of ESP and CBI is stable across sparse test suites of different sizes for the subject programs included in the evaluation. Under the sparsest test suites included in the evaluation, $1/5^{\text{th}}$ of the complete test suite, both ESP and CBI show larger variation in effectiveness 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, Figure 5 reveals that for the subject programs included in the evaluation, if the test suite formed features at least 20 successful (passing) test cases and 10 failing test cases, then the overall size of the test suite has little effect on the $\overline{Cost}$ incurred. For those simulations where a long runtime makes obtaining a high number of test cases impractical, a test suite with 20 successful (passing) test cases and 10 failing test cases is likely to be sufficiently large to apply ESP and get effective results.

5.3. Multiple Faults

The established practice in the statistical debugging community is to evaluate the effectiveness of a debugger by quantifying how well it finds the first fault in a subjectprogram or simulation [Jeffrey et al. 2008; Jones and Harrold 2005]. We followed this practice in the perfect world evaluation to enable ESP to be compared against the leading alternatives. Unfortunately, simulations are frequently written that contain multiple faults. It is important to ensure that ESP has a process for identifying multiple faults and that the process is effective. The following algorithm guarantees that ESP gives at least one failure-predicting predicate for each fault that is present in a simulation: (1) rank each predicate in descending order of suspiciousness, (2) remove the top-ranked predicate p and discard all execution traces where p was true, and (3) repeat the process until either the set of execution traces or predicates is empty.

Five versions of the subject simulations in the perfect world evaluation contain multiple faults. Applying our algorithm to these versions yields results in line with our previous evaluation of ESP. Each fault in each of the five versions was localized with a $\overline{Cost}$ greater than 10%. In the perfect world evaluation the majority of the faults in subject simulation were localized while incurring a $\overline{Cost}$ greater than or equal to 10%. Although we are hesitant to generalize our results based on the small sample of faulty simulation versions and the difficult nature of the multiple faults problem, we are encouraged by ESP's performance.

6. CASE STUDY: ESP IN THE WILD

A case study helps to highlight each of our contributions in combination and elucidate how ESP can be applied for simulation developers attempting to localize a faulty statement causing an unexpected output. Our case study simulation is an agent-based simulation that studies how the availability of different restaurant choices in an area effect obesity. The simulation contains four agent populations: people, homes, restaurants, and workplaces. Inputs include the number of people, the eating habits of the people, the starting age and weight levels of the people, the number of workplaces, the number of restaurants, and the types of restaurants. Every day, each person eats three meals, which are obtained from the restaurants. Each meal contributes a number of calories to the individual based on the restaurant's type. People chose restaurants based on their current location. At the end of every week, each person's weight is adjusted based on the number of calories that they consumed during the week compared to the number of calories that they needed to maintain their weight level. Calorie levels are reset each week, and the obesity of each individual is tracked over time. Agents are removed from the population as a stochastic function of their life expectancy, which is influenced by their obesity.

The simulation is capable of reproducing historical trends in the percentage of (1) obese, (2) overweight, (3) normal weight, and (4) underweight individuals in datasets for different cities in the United States. Each city specifies the restaurant choices that are available, and the datasets measure the percentages of the city population falling into each of the four specified weight categories. However, in some of the cities, individual agents reach abnormally large weights. We refer to these agents as super-gainers. This occurs at a significantly higher rate than it does in the U.S. population. We apply ESP to localize the statement causing this unexpected output.

We use ESP to instrument the simulation source code related to each *person* agent in the simulation to capture elastic and static single variable and scalar pair predicates. The predicates instrumented for each agent are largely composed of seven variables: (1) the *age* of the agent; (2) the *height* of the agent; (3) the *weight* of the agent; (4) the *bmr* of the agent, which is the number of calories the agent's body uses in a day; (5) the *bmi* of the agent, which is a screening measure for obesity; (6) the average number of *calories* the agent consumes in a week; and (7) *lifeExp*, the current life expectancy of the agent given the previously mentioned variables.

We collect the predicate data for each agent at the end of each simulated week. The collected predicates for each agent at the end of each week serve as an execution trace of the simulation. The passing extent, u_{agent}, for the execution trace is based on the agent's weight (*weight*). It is computed using historical data to determine the extent to which the agent's weight drastically deviates (σ_{weight}) from mean weight of the city ($\overline{weight}$). The formula for the many-valued labeling function is shown in Equation (2):

$$u_{agent} = \exp\big(-(weight - \overline{weight})^2/\sigma^2_{weight}\big). \tag{2}$$

To localize the statement causing the unexpected outcome, the simulation is run for one of the cities with a large number of super-gainers. The extent to which an agent is or is not a super-gainer is determined by computing $1 - u_{agent}$. Equation (2) is very similar to the function we used to determine the passing extent of a given execution trace for the `ising` and `sir` models discussed in Section 4.1.2. In general, it reflects a straightforward strategy to determine the extent to which a given value deviates from an expected mean on a [0,1] scale. This strategy is applicable to any stochastic simulation with an expected value and an expected variance.

The most suspicious predicate uncovered by ESP is the elastic scalar pairs predicate: $age_{257} - lifeExp_{271} > \mu_{age-lifeExp} + 3\sigma_{age-lifeExp}$. Recall from the discussion on predicates in Section 3 that this means that agents who have a drastic difference between their age (in line 257) and life expectancy (in line 271) are more likely to be super-gainers.

This predicate localizes the fault. The statement in line 257 of the simulation is poorly written. Under certain conditions, it allows an agent to continue living without checking his or her newly determined life expectancy. Neither the SME or the simulation developer who constructed the simulation realized that such a condition could occur. Given the line of code and the relationship of the variables involved, the fault was quickly located and the simulation's implementation was corrected. In resulting runs, there were significantly fewer super-gainers.

The cause of super-gainers in the model seems clear enough once found. If obese agents are capable of drastically outliving their life expectancy, they will continue to gain weight in perpetuity. Although this bug appears straightforward once identified, it had been present and undiscovered in this simulation for months. Many bugs are obvious only once one knows where to look. ESP directed us to two lines of code and specified a single condition to consider: the difference between the agent's age and the agent's life expectancy is significantly larger than average.

This case study highlights our two contributions working in combination. First, the many-valued labeling function is needed, as the *exact* expected weight of an agent cannot be determined due to the use of stochastic distributions in the simulation. Without the ability to specify Equation (2), it would not be possible to make a distinction between passing and failing execution traces. As a result, effective statistical debugging would not be possible. Furthermore, the condition causing the unexpected condition was isolated by an elastic predicate. The static predicate, $age_{257} - lifeExp_{271} > 0$, identifies when an agent outlives his or her life expectancy. Unfortunately, it is not particularly useful for isolating the fault in the simulation, because it is valid for an agent to briefly outlive his or her life expectancy since agents die off as a stochastic function of the $lifeExp$. However, elastic predicates enable us to realize that when many agents are capable of drastically outliving their life expectancy, super-gainers become rampant.

7. RELATED WORK

ESP is closely related to VV&T. *Verification* reflects confirming that a simulation accurately represents the developer's conceptual model. *Validation* reflects the degree to which a simulation is an accurate representation of the real world from the perspective of intended use. Both definitions stress checking via comparison to a reference standard. However, verification checks that the simulation is solved correctly, whereas validation checks that the simulation matches trusted data or other model outputs [Sargent 2013]. Testing is inherent in simulation verification and validation. *Testing* compares an expected result against model output. When testing uncovers a model output that deviates from the expected result, ESP finds the source code statement causing the deviation. Given the complementary relationship of ESP and VV&T, we review work related to both.

Simulation verification techniques depend on simulation implementation languages. Simulation-specific languages (Arena, Simulink) entail verification via structured walk-throughs of the code and program traces that are tested against expected results. These techniques check that the language, the pseudo–random number generator, and the simulation are correct [Sargent 2013]. Implementing a simulation in a general-purpose programming language (Java, C/C++) allows for model checking to be employed. Model checking ensures that each requirement specified in the design of the simulation is satisfied in the implementation [Bérard et al. 2010]. This analysis is powerful, but model checking tools often have a steep learning curve.

Simulation validation techniques are both subjective and objective. Subjective ones rely on the judgment of SMEs, whereas objective techniques use statistical tests. Subjective techniques include (1) *animation* and *operational graphics* to test that simulation output dynamics are behaving correctly over time [Rohrer 2000] and (2) *face validity* and *turing tests* where SMEs assess if the simulation output is a reasonable facsimile of the real system [Colby et al. 1972]. Objective tests include (1) *exploratory and sensitivity analysis* to statistically test if the simulation inputs have the same effect on the outputs as they do in the real system [Saltelli et al. 2000; Birta and Özmizrak 1996; Barton and Lee 2002] (2) and *stability analysis* to quantify the variance of a simulation output [Higham 2001; Srivastava et al. 2002].

8. CONCLUSIONS

Although statistical debuggers are generally effective, they are not tailored to models and simulations employing continuous numbers and random variates. Elastic predicates and many-valued labeling functions address that deficiency. Elastic predicates complement the static predicates employed in existing statistical debuggers by profiling variable values to create partitions that isolate faulty ranges of values. Many-valued labeling functions maximize the effectiveness of elastic predicates by creating a process to differentiate between execution traces that produce the specified output (and pass) versus those execution traces that produce unspecified output (and fail) in the face of stochastics. These two contributions both hold in an imperfect world.

REFERENCES

George K. Baah, Andy Podgurski, and Mary Jean Harrold. 2011. Mitigating the confounding effects of program dependences for effective fault localization. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*. ACM, New York, NY, 146–156. DOI:http://dx.doi.org/10.1145/2025113.2025136

Paul I. Barton and Cha Kun Lee. 2002. Modeling, simulation, sensitivity analysis, and optimization of hybrid systems. *ACM Transactions on Modeling and Computer Simulation* 12, 4, 256–289.

Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. 2010. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer.

Louis G. Birta and F. Nur Özmizrak. 1996. A knowledge-based approach for the validation of simulation models: The foundation. *ACM Transactions on Modeling and Computer Simulation* 6, 1, 76–98.

Phelim P. Boyle and Sok H. Lau. 1994. Bumping up against the barrier with the binomial method. *Journal of Derivatives* 1, 4, 6–14.

Guillaume Brat, Klaus Havelund, SeungJoon Park, and Willem Visser. 2000. Model checking programs. In *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE)*. 3–12.

Stephen G. Brush. 1967. History of the Lenz-Ising model. *Reviews of Modern Physics* 39, 4, 883–893. DOI:http://dx.doi.org/10.1103/RevModPhys.39.883

Wlodzimierz Bryc. 1995. *The Normal Distribution: Characterizations with Applications*. Springer-Verlag. http://books.google.com/books?id=BQ7vAAAAMAAJ.

Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. 2009. HOLMES: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE, Los Alamitos, CA, 34–44. DOI:http://dx.doi.org/10.1109/ICSE.2009.5070506

Holger Cleve and Andreas Zeller. 2005. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*. ACM, New York, NY, 342–351. DOI:http://dx.doi.org/10.1145/1062455.1062522

Kenneth Mark Colby, Franklin Dennis Hilf, Sylvia Weber, and Helena C. Kraemer. 1972. Turing-like indistinguishability tests for the validation of a computer simulation of paranoid processes. *Artificial Intelligence* 3, 199–221.

Kai Detlefsen and Wolfgang K . Hardle. 2007. Calibration risk for exotic options. *Journal of Derivatives* 14, 4, 47–63.

Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Vol. 1. Prentice Hall, Englewood Cliffs, NJ.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2, 99–123. DOI:http://dx.doi.org/10.1109/32.908957

Theodore J. Gordon. 2003. A simple agent model of an epidemic. *Technological Forecasting and Social Change* 70, 5, 397–417.

Ross Gore. 2012. *Fault Localization for Exploratory Simulations*. Ph.D. Dissertation. University of Virginia, Charlottesville, VA.

Donald Gross, John F. Shortle, James M. Thompson, and Carl M. Harris. 2011. *Fundamentals of Queueing Theory*. John Wiley and Sons. http://books.google.com/books?id=K3lQGeCtAJgC.

Desmond J. Higham. 2001. An algorithmic introduction to numerical simulation of stochastic differential equations. *SIAM Review* 43, 3, 525–546.

Teerawat Issariyakul and Ekram Hossain. 2011. *Introduction to Network Simulator NS2*. Springer.

Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. 2008. Fault localization using value replacement. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08)*. ACM, New York, NY, 167–178. DOI:http://dx.doi.org/10.1145/1390630.1390652

James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*. ACM, New York, NY, 273–282. DOI:http://dx.doi.org/10.1145/1101908.1101949

W. David Kelton. 2007. Representing and generating uncertainty effectively. In *Proceedings of the 39th Conference on Winter Simulation: 40 years! The Best Is Yet to Come (WSC'07)*. IEEE, Los Alamitos, CA, 38–42. http://dl.acm.org/citation.cfm?id=1351542.1351552

Donald E. Knuth. 1997. *The Art of Computer Programming: Seminumerical Algorithms*. Vol. 2 (3rd ed.). Addison Wesley Longman, Boston, MA.

David Krahl. 2005. Debugging simulation models. In *Proceedings of the Winter Simulation Conference*. IEEE, Los Alamitos, CA, 7.

Averill M. Law and W. David Kelton. 1991. *Simulation Modeling and Analysis*. Vol. 2. McGraw-Hill, New York, NY.

Ben Liblit. 2008. Cooperative debugging with five hundred million test cases. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08)*. ACM, New York, NY, 109–120. DOI:http://dx.doi.org/10.1145/1390630.1390632

Sergei Mikhailov and Ulrich Nögel. 2003. Heston’s stochastic volatility model: Implementation, calibration, and some extensions. *Wilmott Magazine* 4, 74–79.

Gonzalo Navarro. 2001. A guided tour to approximate string matching. *ACM Computing Surveys* 33, 1, 31–88. DOI:http://dx.doi.org/10.1145/375360.375365

Matthew W. Rohrer. 2000. Seeing is believing: The importance of visualization in manufacturing simulation. In *Proceedings of the 32nd Conference on Winter Simulation*. 1211–1216.

Francisco J. Ros. 2007. UM-OLSR, an implementation of the OLSR (Optimized Link State Routing) protocol for the ns-2 network simulator. Retrieved March 6, 2015, from http://masimum.inf.um.es/fjrm/development/um-olsr/.

Andrea Saltelli, Karen Chan, and E. Marian Scott. 2000. *Sensitivity Analysis*. Vol. 134. Wiley, New York, NY.

Robert G. Sargent. 2013. Verification and validation of simulation models. *Journal of Simulation* 7, 1, 12–24.

Ranjan Srivastava, Lingchong You, J. Summers, and John Yin. 2002. Stochastic vs. deterministic modeling of intracellular viral kinetics. *Journal of Theoretical Biology* 218, 3, 309–321.

Received February 2013; revised August 2014; accepted November 2014