Proceedings of the 2010 Winter Simulation Conference
B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, eds.

IMPROVED METHODS AND MEASURES FOR COMPUTING DYNAMIC PROGRAM SLICES IN STOCHASTIC SIMULATIONS

Ross Gore
Paul F. Reynolds, Jr.

University of Virginia
151 Engineer's Way
P.O. Box 400740
Charlottesville, VA, 22901 USA

ABSTRACT

Stochastic simulations frequently exhibit behaviors that are difficult to recreate and analyze, owing largely to the stochastics themselves, and consequent program dependency chains that can defy human reasoning capabilities. We present a novel approach called Markov Chain Execution Traces (MCETs) for efficiently representing sampled stochastic simulation execution traces and ultimately driving semi-automated analysis methods that require accurate, efficiently generated candidate execution traces. The MCET approach is evaluated, using new and established measures, against both additional novel and existing approaches for computing dynamic program slices in stochastic simulations. MCET's superior performance is established. Finally, a description of how users can apply MCETs to their own stochastic simulations and a discussion of the new analyses MCETs can enable are presented.

1 INTRODUCTION

The daunting nature of quantifying, analyzing and understanding uncertainty in model design and simulation outcomes is evident in the results of epidemiology studies conducted this century. Epidemiologists have addressed the question of government level actions and reactions regarding the spread of infectious diseases such as smallpox and bird flu. Should a comprehensive vaccination program be initiated? How and to what degree should infected individuals be isolated, and for how long? The range of answers to these questions is broad and full of conflict. Recently, Elderd, Dukic and Dwyer (2006) have shown analytically that just four of the potentially hundreds of critical independent variables in these studies induce extreme sensitivity in model predictions, leading to serious conflict regarding remedial approaches involving billions of dollars and millions of people. Subject matter experts (SMEs) must be given additional capabilities to understand the behavior of their simulations so that results can be used effectively and with confidence.

Exploring a behavior in a simulation with uncertainty - a *stochastic simulation* - is typically time consuming and mostly manual. Generally the objective of an exploration is to identify the program statement(s) that cause a simulation behavior of interest. Thus, any improvements in this search process can greatly decrease the cost of exploration.

In traditional exploration techniques users apply debuggers, e.g., the Eclipse debugger (Eclipse Foundation 2010), partially automated debuggers (Duccasse 1999), and fault localization tools (Jones and Harrold 2005) to identify program statements in the simulation source code that lead to the behavior of interest. These techniques rely on program slicing to identify the program statements that could impact the

behavior under exploration. Unfortunately, the program slicing methods employed in these tools are not the best choice for stochastic simulation analysis, as we argue next.

Program slicing is a decomposition technique that extracts statements *relevant* to a particular computation within the program or simulation (Weiser 1984). A program slice provides the answer to the question, "What program statements affect the computation of variable *v* in line number *l*?" (Binkley and Gallagher 1996). In a program *x*, a program statement *s* is *relevant* to a variable *v* in line number *l* if *s* lies on the path, according to data and control flow dependences, to the computation of *v* in line number *l.* For a program *x*, that employs random variables (a stochastic program), and a specified input *i*, it is possible that for some executions *x(i)*, *s* will be relevant to the computation of *v* in line number *l* and for other executions *x(i), s* will not be relevant to the computation of *v* in line number *l*. Current methods and measures of relevance are unable to capture this variability. This is the *stochastic simulation – dynamic program slicing problem*.

In order to accurately and realistically evaluate the effectiveness of candidate solutions to the stochastic simulation – dynamic program slicing problem, we present two new measures: *cumulative relevance* and *expected relevance*. Also, we present several different methods to address the stochastic simulation – program slicing problem and evaluate them, using our two new measures as well as the established measure of efficiency, all for three published stochastic simulations.

The most promising method, by our measures, is one first presented here: Markov Chain Execution Traces (MCETs). MCETs form a Markov Chain model of a sample of execution traces for a stochastic simulation given a specified input. Once an MCET is created a trajectory through that MCET can be simulated yielding an execution trace that is *representative* of the observed executions. The resulting trace is *representative* in the sense that: 1) every program statement that occurs in the samples has the possibility of being included, 2) the order and the control-flow structure of the statements in the trace reflect that of the samples and 3) the trace is likely to contain program statements and control flow structures from most of samples.

Work related to the MCETs and program slicing is presented in Section 2. Section 3 provides the details of how MCETs are constructed and how a trajectory through the MCET is simulated to create a representative execution trace. In Section 4, the details of established and new methods to address the dynamic program slicing – stochastic simulation problem and the two new evaluation measurers are presented. Section 5 describes the evaluation of these methods for three published stochastic simulations. Finally, in Section 6, we summarize our contributions and discuss future work.

2 RELATED WORK

MCETs draw on the areas of Markov chains, software testing and program slicing. In this section we review work in each of these areas and describe how it relates to MCETs.

2.1 Markov Chains

A Markov chain is composed of a set of states, $S = \{s_1, s_2 ... s_n\}$ where $|S| = n$ and *P*, a *n* x *n* transition probability matrix, $P = \begin{bmatrix} p_{11} \cdots p_{1n} \\ p_{i1}\, p_{ij}\, p_{in} \\ p_{n1} \cdots p_{nn} \end{bmatrix}$. Each entry, p_{ij}, in *P* represents the probability of moving from the current state s_i to the next state s_j. Each move is called a step. The probability of stepping from s_i to s_j depends only on the current state s_i, not on any previous state (Meyn and Tweedie 2009). Markov chains have been successfully applied in modeling and simulation for likelihood ratio gradient estimation (Glynn 1987), simulation optimization (Olafsson and Shi 1999) and rare-event simulation (Heidelberger 1995). One of the applications that closely resembles MCETs is the use of Markov Chains to generate

"realistic" or "representative" text documents (Kenner and Joseph 1984, Hartman 1996). MCETs employ Markov chains in a similar manner to generate representative execution traces of stochastic simulations.

2.2 Statistical Software Testing

Markov Chains have also been shown to be a valuable resource in the statistical software testing community. It is impossible to test most modern software systems completely because the application of every input combination or scenario (exhaustive testing) is often infeasible. The implication for testing is that only a small subset of the possible input space can be exercised before release. As a result statistical testing is used to explore software functionality (Whittaker and Thomason 2000). In statistical testing of software all possible uses of the software, at some level of abstraction, are represented by a statistical model. The application of Markov Chains as the statistical model enables the generation of test cases that are representative of real-world use cases once the software is deployed. Within the statistical software testing community the application of Markov Chains has enabled several new analyses (Whittaker and Thomason 2000). We expect MCET will create similar opportunities within the stochastic simulation community. These opportunities are discussed as opportunities for future work in Section 6.

2.3 Static and Dynamic Program Slicing

Program slicing is a decomposition technique that extracts statements relevant to a particular computation within the program (Weiser 1984). An important distinction is that between static and dynamic slices. Figure 1 (a) shows an example program that reads an integer input n, and computes the sum and the average of the first n positive numbers. If the sum of the first n integers is evenly divisible by n the program assigns -1 to x. Otherwise the program assigns sum to x. The criterion for a static slice is a 2-tuple consisting of {line number of statement *s*, the name of variable *v*}, where *v* is the variable of interest and *s* is the statement of interest. Figure 1 (b) shows a static slice of this program using criterion {13, x}. Slices are computed by identifying consecutive sets of transitively relevant statements, according to data and control flow dependences (Tip 1995). Only statically available information is used for computing slices; hence, this type of slice is referred to as a *static* slice.

```
1    read(n);                    1    read(n);                    1    read(n);
2    i := 1;                     2    i := 1;                     2    i := 1;
3    x := 0;                     3    x := 0;                     3    x := 0;
4    sum := 0;                   4    sum := 0;                   4    sum := 0;
5    average := 0;
6    while i<= n                 6    while i<= n                 6    while i<= n
7       sum := sum + i;          7       sum := sum + i;          7       sum := sum + i;
8       i := i + 1;              8       i := i + 1;              8       i := i + 1;
9    end                         9    end                         9    end
10   if (sum mod n == 0)         10   if (sum mod n == 0)         10 // (sum mod n == 0)
11      x := -1;                 11      x := -1;                 11     x := -1;
     else                             else
12      x := sum;                12      x := sum;
13   print (x);                  13   print (x);                  13   print (x);
14   average := sum/n;
15   print (average);
```

Figure 1(a): a) An example program. (b) A static slice of the program using criterion {13 ,x}. (c) A dynamic slice of the program using criterion {n = 4, 13, x}.

In the case of dynamic program slicing, only the dependences that occur in a specific execution of the program are taken into account. A dynamic slicing criterion specifies the input; it consists of {input, line number of statement *s*, name of variable *v*}. The difference between static and dynamic slicing is that

dynamic slicing assumes fixed input for a program, whereas static slicing does not make assumptions regarding the input. Figure 1(c) shows a dynamic slice of the program in Figure 1(a) using the criterion {n = 4, 13, x}. Note that for input n = 4, the assignment x := sum is executed, and the assignment x := -1 is not executed. The "if (sum mod n = 0)" branch of statement 9, and statement 10 in Figure 1(a) is omitted from the dynamic slice because the assignment of x := -1 is not executed.

It is beneficial to distinguish between a dynamic program slice and an execution trace. An execution trace includes each execution of a statement, in the order the statements are executed, for a specified input to the program *i*. Within an execution trace most statements are included multiple times since most statements in a program are executed multiple times. In contrast, a dynamic program slice only contains one entry for each statement in the program that is relevant to the computation specified by the dynamic program slicing criterion. A dynamic program slice can be described as the intersection of a static program slice given the slicing criterion and an execution trace for the input specified in the dynamic program slicing criterion.

Previous researchers have used static and dynamic program slicing separately and in combination to enable program debugging. In debugging, one is often interested in a specific execution of a program that exhibits anomalous behavior, which in part matches our goal of exploring the behavior of a stochastic simulation. Dynamic slices are particularly useful here, because they only reflect the actual dependencies of that execution, resulting in smaller slices than static ones (Korel and Rilling 1997).

3 MARKOV CHAIN EXECUTION TRACES

When a behavior is first observed in a stochastic simulation, the prospect of exploring and then explaining that behavior can be daunting. Most users apply debugging techniques mentioned earlier, such as classic debuggers (Eclipse Foundation 2010), partially automated debuggers (Duccasse 1999), and fault localization tools (Jones and Harrold 2005) to identify program statements that lead to the behavior. The process is time-consuming and cumbersome due to the mismatch of the stochastic behaviors of simulations and the assumption of deterministic behavior in the existing tools. We realized that a research artifact that could automatically generate execution traces that were *representative* of an observed sample of executions was needed. Thus, Markov chain execution traces (MCETs) were born. MCETs enable an improved solution to the stochastic simulation – dynamic program slicing problem when evaluated against new and existing methods. The MCET approach is not computationally intensive because the number of execution trace samples required to create a MCET is relatively small. Efficiency of the MCET approach to computing a dynamic program slice compared to existing approaches is evaluated and discussed in Section 5.

3.1 Generating a Markov Chain of Execution Traces

The method for generating MCETs is a straightforward application of modeling a sample of execution traces with Markov Chains. Generating a MCET assumes a stochastic simulation, an input *i* to the simulation and an integer *n*, where *n* specifies the number of execution traces samples to generate. Next, the stochastic simulation is executed *n* times with input *i*. For each execution, the execution trace is stored. Once all *n* executions are completed and the corresponding execution traces are stored, the Markov chain is formed. Recall a Markov chain is a set of *n* states, *S*, and a *n* x *n* matrix of transition probabilities, *P*. Each unique statement in the observed execution traces is a state in the Markov chain. For example, the program in Figure 1 would result in at most 15 states in the Markov Chain because there are only 15 unique statements in the program. Next, the probability matrix is created using the sequence of statements in the *n* observed execution traces. Each entry, p_{ij}, in the matrix *P* represents the frequency with which the statement represented by state s_j immediately follows the statement represented by state s_i in the observed execution traces. The Markov chain of the sampled execution traces has the following properties:

1. Each statement observed in the sample of execution traces is represented.
2. States transition from one to another with the same probability observed in the sample of execution traces.

3. The transition from one state, s_i, to another state, s_j only depends on s_i, not any other state.

Next, we describe simulating a trajectory through the MCET to generate *representative* execution traces.

3.2 Simulating a Trajectory through the MCET

Assuming the construction of the MCET, simulating a trajectory as described in the following steps results in a representative execution trace (RET).

1. The median length, l, of the n sampled execution traces is computed.
2. The trajectory begins at the start state of the Markov chain. The start state is the statement which begins each execution trace of the observed samples. The statement represented by the start state is added to RET.
3. The next state, s_j, is chosen at random from the states connected to the current state, s_i, using the probability transitions in matrix P. The statement represented by s_j is added to RET.
4. Step 3 is repeated until $|\text{RET}| = l$ or an end state, an s_i with no outgoing edges is reached.

The resulting RET is referred to as *representative* due to its length and the three properties of the MCET driving the creation of the RET. The properties guarantee that every statement in the observed execution traces can be included in the RET. Furthermore, due to the nature of the transition probabilities in the MCET the order of statements and control-flow structure in the RET reflects the order of statements and structure in the observed execution traces. Finally, statements from most of the observed execution traces are included in the RET because each transition in the generating process only depends on the current statement. Because the length of the RET is bounded by l, the algorithm terminates and the RET has the median length of the observed traces or reflects the termination of an observed trace.

4 MEASURES AND METHODS

We begin by presenting new measures for evaluating the effectiveness of approaches to the stochastic simulation – dynamic program slicing problem. Following that, we present alternative methods for addressing the problem, including one that uses MCETs and RETs.

4.1 Measures for the Stochastic Simulation – Dynamic Program Slicing Problem

The program in Figure 2(a) highlights the stochastic simulation – dynamic program slicing problem. The program is not meant to be representative of a real stochastic simulation, but to demonstrate the problem.

```
1     read(n);
2     x := 0;
3     rand := randNum(0, 1);
4     if (rand >= .998 &&
          rand <= .999)
5         x := rand +n;
      else
6         x := n;
7     print(x);
```

```
1     read(n);
2     x := 0;

//    (rand >= .998 &&
//     rand <= .999) == false


6         x := n;
7     print(x);
```

```
1     read(n);
2     x := 0;
3     rand := randNum(0, 1);
//    (rand >= .998 &&
//     rand <= .999) == true
5         x := rand +n;
      else

7     print(x);
```

Figure 2: (a) A stochastic program. (b) One possible dynamic slice of the program using criterion {n=13,7,x}. (c) Another possible dynamic slice of the program using the same criterion.

From the user's point of view the behavior of the example program in Figure 2(a) is stochastic. Figures 2(b) and 2(c) show the two possible dynamic program slices using criterion {n = 13, 7, x} for the

program in Figure 2(a). Figure 2(b) shows the dynamic program slice, when the random number generator does not generate a number between .998 and .999. Figure 2(c) shows the dynamic program slice when the random number generator does generate a random number between .998 and .999. Assuming a uniform random number generator the dynamic program slice for the program in Figure 2(a) is the program slice shown in Figure 2(b) approximately 99.9% of the time and the program slice shown in Figure 2(c) 0.1% of the time. When 2(b) is executed, statements 1, 2, 4, and 6 are relevant to the output value of *x* in statement 7. However, when 2(c) is executed, statements 1, 2, 3, 4 and 5 are relevant to the output value of *x* in statement 7. Current methods and measures of relevance are unable to capture this.

A continuous measure of the relevance of a program statement *s* to a variable *v* in line number *l* can be proposed to evaluate how well new and existing approaches address this problem. Each execution, *x(i)*, of a stochastic simulation can be considered a Bernoulli trial *Y* such that: if *s* is relevant to the computation of *v* in line number *l* in the execution *x(i)*, *Y =1* and if *s* is not relevant to the computation of *v* in line number *l, Y=0*. Thus for each execution *x(i)*, *s* is relevant to the computation of *v* in line number *l* with probability *p*. For a set of *N* executions an unbiased estimator $\hat{p}$, of the probability that statement *s* is relevant to the computation of *v* in line number *l* for program *x* with input *i* $\frac{\#\text{ of } x(i) \text{ where } s \text{ is relevant to } v \text{ in line number } l}{N}$. Let *T* be the complete set of statements in the source code of program *x*, the *total relevance* in *x(i)*, given *v* and *l* is the sum of the relevance, $\hat{p}_k$, of all the statements $t_k \in$ T; symbolically this is: $\sum_{k=1}^{|T|} \hat{p}_k$.

Using the continuous definition of relevance and the definition of total relevance for a program *x*, given an *i*, *v* and *l*, *cumulative relevance* and *expected relevance* can be defined. Let *S* be the set of statements returned by the slicing tool, s_j be a statement in *S* and $\hat{p}_j$ be the estimate of the probability that s_j is relevant to the computation of interest. The ***cumulative relevance*** of *S* is the sum of the $\hat{p}_j$ of all $s_j \in S$ divided by the total relevance for the program ; symbolically the cumulative relevance is: $\sum_{j=1}^{|S|} \hat{p}_j \Big/ \sum_{k=1}^{|T|} \hat{p}_k$.

The ***expected relevance*** of *S* is the sum of the $\hat{p}_j$ of all $s_j \in S$; divided by the cardinality of S; symbolically the expected relevance is: $\sum_{j=1}^{|S|} \hat{p}_j \Big/ |S|$.

Cumulative and expected relevance are strongly related to the recall and precision measures employed in information retrieval method evaluations (Baeta-Yates and Riberio - Neto 1999). In the information retrieval community a recall score of 1.0 means that the set of search results includes all the results relevant to a query, however, recall does not measure how many of the search results were not relevant to the query. Similarly, a cumulative relevance score of 1.0 means all the statements in the program *x* that are relevant to the behavior are included in *S*, but does not measure how many statements included in *S* are not relevant to the behavior.

A precision score of 1.0 means that every result in the set of returned search results is relevant to the search query, however, precision does not measure how many relevant search results were not returned. Similarly, an expected relevance score of 1.0 means each statement included in *S* is relevant to the behavior but does not measure how many relevant statements were not included in *S* (Baeta-Yates and Riberio – Neto 1999).

These measures will be used to define effectiveness in the evaluation of the new and existing approaches to the stochastic simulation – dynamic program slicing problem for three published stochastic simulations. Next, these approaches and how each relates to the program in Figure 2 are presented.

4.2 Static Program Slice Method

The static program slice method is a commonly used approach to dynamic program slicing employed in debugging tools. The method returns the set of statements *S* that are included in a static program slice of the simulation. Given the program in Figure 2, a static program slice would result in statements {1, 2, 3, 4, 5, 6} being included in the set of statements, *S*. The method is implemented using the Kaveri static program slicing tool (Jayaraman, Ranganath and Hatcliff 2005).

4.3 Single Dynamic Program Slice

The single dynamic slice method is another commonly used approach to dynamic program slicing employed in existing debugging tools. The method returns the set of statements *S* that are included in a single dynamic program slice of the simulation. Given the program in Figure 2, a single dynamic program slice would result in statements {1, 2, 4, 6} being returned if `rand` is not between .998 and .999 or the statements {1, 2, 3, 4, 5} if `rand` is between .998 and .999 being returned. The method is implemented using the JSlice dynamic program slicing tool (Wang 2007).

4.4 Set-union of Dynamic Program Slices Method

The set-union of dynamic program slices method employs *n* execution traces of the simulation for a specified input *i*. The set of execution traces is: $ET = \{et_1,...et_i,...et_n\}$. A static program slice of the simulation, *SPS*, is also computed. For each execution trace in ET, et_i, the set of statements common to et_i and *SPS* is computed; this is dps_i, the dynamic program slice for et_i. The result is, $DPS = \{dps_1,...,dps_i,...dps_n\}$, a set of dynamic program slices that corresponds to each respective entry in ET. The set-union of dynamic program slices method computes a set *S* by removing the union of all unique program statements included in *DPS*. Given the program in Figure 2, the method returns statements {1, 2, 3, 4, 5, 6}. It is implemented using the JDI libraries (Java 2010) and the Kaveri static program slicing tool (Jayaraman, Ranganath and Hatcliff 2005).

4.5 Set-intersection of Dynamic Program Slices

The set-intersection of dynamic program slices method employs *n* execution traces of the simulation for a specified input *i*. The set of execution traces is: $ET = \{et_1,...et_i,...et_n\}$. A static program slice of the simulation, *SPS*, is also computed. For each execution trace in ET, et_i, the set of statements common to et_i and *SPS* is computed; this is dps_i, the dynamic program slice for et_i. The result is $DPS = \{dps_1,...,dps_i,...dps_n\}$, a set of dynamic program slices that corresponds to each respective entry in ET. The set-intersection of dynamic program slices method computes a set *S* by removing the intersection of all program statements in *DPS*. Given the program in Figure 2, the method returns statements {1, 2, 4}. It is implemented using the JDI libraries (Java 2010) and the Kaveri static program slicing tool (Jayaraman, Ranganath and Hatcliff 2005).

4.6 Markov Chain Execution Trace (MCET) Method

The MCET method employs *n* execution traces of simulation for a specified input *i* to form a MCET and generate a RET. The MCET Method extracts each unique statement from the RET and includes these statements in the set of statements, *S*, returned by the method. Given the program in Figure 2, the method

could return several sets of statements including {1,2,4,6}, {1,2,3,4,6}, {1,2,3,4,5} and {1,2,4,5}. It is implemented using the Java JDI libraries (Java 2010), Kaveri static program slicing tool and the Indus static analysis libraries (Jayaraman, Ranganath and Hatcliff 2005).

5 EVALUATION

In order to evaluate the effectiveness and efficiency of the approaches in Section 4 to the stochastic simulation – dynamic program slice problem we performed an evaluation using three published stochastic simulations. This section describes the evaluation and presents the results and the analysis of it.

5.1 The Dunham SEIR Disease Spread Simulation

The Dunham simulation predicts disease spread by modeling interactions on a 2-D torus. At each time step, infectious individuals in proximity to susceptible individuals within a specified radius spread their infection with a given probability (Dunham 2005). The predictions of the simulation for a population of size 100 for a period of 100 days is evaluated.

5.2 Queueing Simulation

Examples from queueing theory are pervasive throughout modeling and simulation (Law and Kelton 2000). The following queueing simulation is used in the evaluation: A finite population of customers arrive to a set of first in, first out (FIFO) service stations. Each service station is equipped with a queue. The arrival rate of each customer and the assignment of a customer to a service station are uniformly distributed. Each customer is also assigned a uniformly distributed required service time. If too many customers are in the queue to which a customer is assigned, the customer may choose not to join the queue and leave without being serviced. If the customer has already joined the queue but has been waiting too long for service, the customer may choose to leave the queue without service or join the queue for another service station. Given this model the average wait time for 100 customers is evaluated (Allen 1990).

5.3 Self-Driven Particle Simulation

In the self-driven particle simulation particles interact on a 2-dimensional torus according to a simple rule. Particles move at a constant speed, and their orientation is set to be the average orientation of all particles within an interaction radius plus a random term. Under most parameterizations particles form clusters given the set of rules. The median number of particles in a cluster is evaluated (Phet 2010).

5.4 The Variables and Measures

The evaluation manipulated one independent variable: the method used to compute the dynamic program slice. Each of these methods is described in Section 4. To compare these methods we use two dependent variables: effectiveness and efficiency. To evaluate effectiveness we use the harmonic mean, or F_1 measure of cumulative relevance and expected relevance. The F_1 measure equally weights expected relevance and cumulative relevance and is commonly used in the information retrieval community to combine recall and precision (Baeta-Yates and Ribeiro-Neto 1999). Formally, the F_1 measure is: $2 * \frac{(cumulative\ relevance\ *\ expected\ relevance)}{(cumulative\ relevance\ +\ expected\ relevance)}$. To evaluate efficiency we recorded the timing of each presented method. Time is measured in wall clock seconds and includes both computation and I/O.

5.5 Experimental Design and Analysis Strategy

Each method is run for each stochastic simulation yielding a set of statements *S* representing the dynamic program slice for the behavior of interest. For the three methods that employ a set of execution traces to

compute S, n=10 execution traces are used. The expected and cumulative relevance for the S returned by each method is calculated using a *monte-carlo function*. The monte-carlo function is constructed by running each stochastic simulation many times and computing the frequency with which each statement in the simulation source code is relevant to the behavior of interest. The Monte-Carlo function appeals to the law of large numbers to accurately compute the relevance of each statement in the source code (Berg 2004). For this evaluation each simulation was run 1,000 times to create the monte-carlo function. The number 1,000 was chosen due to our familiarity with the structure of the simulation and the relevance of the statements within the simulation. Due to the existence of pathological examples it is impossible to specify a number large enough to guarantee proper construction of a monte-carlo function for all stochastic simulations. However, based on stochastic simulations we have observed in practice most monte-carlo functions can be constructed using ≤10,000 simulations runs.

5.6 The Evaluation Results and Analysis

The results of the evaluation concerning the effectiveness dependent variable are presented in Table 1 and Figure 3. In Table 1 the cumulative relevance and expected relevance for each approach, for each stochastic simulation is recorded. The results in Figure 3 depict the F_1 measure computed from the data in Table 1. Overall, Figure 3 shows that the static program slice method is the least effective. This is expected. The static program slice method employs conservative analysis resulting in a set of statements that has perfect cumulative but poor expected relevance. This is due to the inclusion of many statements that while relevant to the behavior of interest for some input are not relevant to the behavior for the specified input.

Table 1: The cumulative and expected relevance of each method for the stochastic simulations.

Cumulative Relevance, Expected Relevance	**Dunham SEIR**	**Queueing Sim.**	**S.D. Particle**
MCET method	0.9886, 0.9475	0.9487, 0.9507	0.9633, 0.9541
Set-intersection dynamic program slice method	0.9306, 1.0000	0.6076, 1.0000	0.9028, 1.0000
Set-union dynamic program slice method	1.0000, 0.9045	0.9990, 0.9031	0.9851, 0.7591
Dynamic program slice method	0.9334, 0.9999	0.7688, 0.9812	0.9013, 0.9987
Static program slice method	1.0000, 0.4136	1.0000, 0.7053	1.0000, 0.4597

For two of the three simulations (The Dunahm SEIR and Self-driven Particle) both the single dynamic program slice and the set-intersection dynamic program slice method perform well. This is due to a path in the simulation that is followed with high probability and touches most of the statements relevant to the behavior of interest. As a result the single dynamic program slice and the set-intersection dynamic program slice methods are likely to capture the statements along this high probability path. Since these statements are along a high probability path, the expected relevance of these statements is high. Also, since the path touches most of the relevant statements in the source code, the cumulative relevance is high. However, when a simulation contains several equally likely paths, as the Queueing simulation does, the dynamic program slice and set-intersection dynamic program slice method cannot have high cumulative relevance. Instead, the set-union dynamic program slice method is likely to uniformly sample all paths, yielding a set of statements with high cumulative and expected relevance.

The MCET method is the most effective approach for each of the simulations. The success of the MCET Method is due to the use of Markov Chains to generate representative execution traces that balance the approaches used in the set-intersection dynamic program slice method and the set union dynamic program slice method. By generating representative execution traces the MCET method is able to include statements with high relevance from the different sampled executions. These properties are reflected through the cumulative and expected relevance measures of the MCET method for each of the three simulations. While the MCET method does not outperform the other methods in either measure, the evenly weighted harmonic mean of the two measures, the F_1 measure, outperforms all the other methods. The complete efficiency results for the three stochastic simulations can be found in (Gore 2010). In summary,

the dynamic program slice method is the most efficient method, followed by the static program slice method which is ~5x slower. The use of sampling decreases the efficiency of the three new approaches causing each to be ~10x slower. Given these results the dynamic program slice method appears to be an effective and efficient solution to the dynamic program slicing – stochastic simulation problem. The method's effectiveness closely approaches the effectiveness of the leading solution, the MCET method for two of the three simulation and it is ~10x faster than the MCET method. However, the evaluation results of the dynamic program slicing method are misleading.

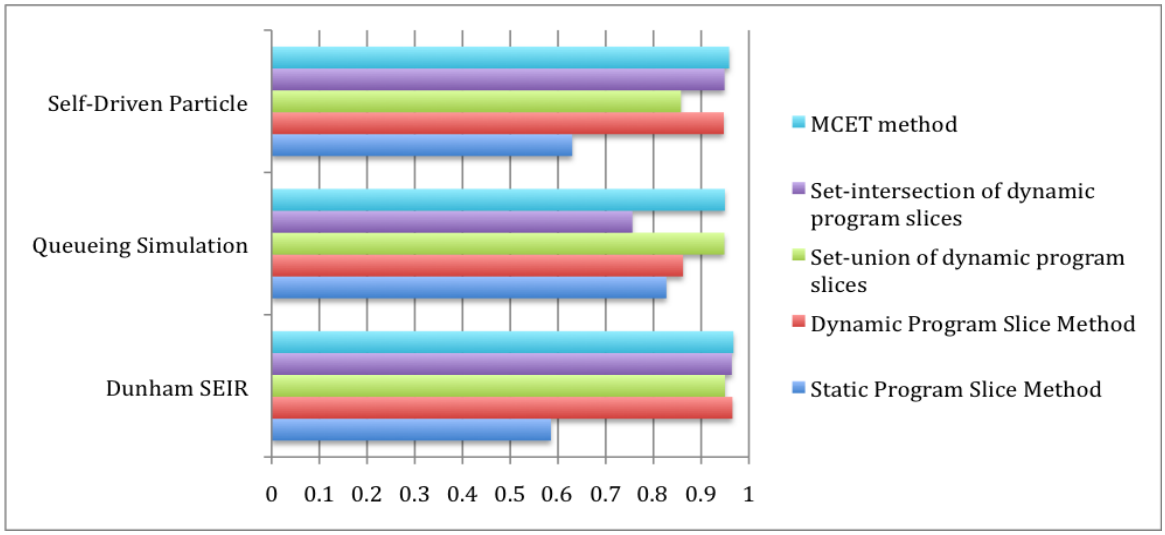

Figure 3: The F_1 measure of each method for the stochastic simulations.

First, the choice of stochastic simulations favors the dynamic program slice method. As discussed earlier, two of the three simulations in the evaluation contain a path that is followed with high probability. As a result, a single dynamic program slice is likely to capture statements along that path yielding a very effective, very efficient solution. However, if more simulations without high probability paths, simulations similar to the Queueing simulation, had been included the dynamic program slice method would appear less effective. While the method works well for simulations with high probability paths it is not an effective general purpose approach to computing dynamic program slices for stochastic simulations.

Second, the evaluation of efficiency only includes computation and I/O time. It does not include the *user time*, the amount of time a user would spend using the resulting set either directly or indirectly (through the use of a tool employing the approach). Since the MCET method computes the most effective dynamic program slice for each simulation it is the only tool that minimizes user time. The tradeoff between computation and I/O time and user time is well established in the simulation community. Most simulation optimization algorithms require exponential computational time to minimize or maximize a given function but require minimal user time (Fu 2002). This property makes these algorithms effective and popular in practice. We expect it to make the MCET method effective and popular as well.

Furthermore, the decrease in efficiency for the MCET method is bounded by the number of samples, n, used in the method. A user (or tool) that employs the methods can reduce n to improve efficiency and still gain some effectiveness over existing approaches. Given this reasoning, the MCET is the leading approach to computing dynamic program slices in stochastic simulations.

6 CONCLUSION

Simulation has become the tool of scientific analysis under circumstances where it is infeasible or impractical to study a system directly (Whipple 1996, Arthur 1999, Elderd, Dukic and Dwyer 2006). Everyday policy debates involving stochastic simulations raise the perfectly legitimate question of whether decision

makers can use simulation-based predictions with confidence. How can policy makers make informed decisions involving billions of dollars and millions of people in confidence when methods, measures and tools to explore these predictions are lacking? This need has motivated the design and development of MCETs.

MCETs generate an execution trace of a stochastic simulation that is representative of a group of sampled execution traces. The execution trace is *representative* in the sense that: 1) every program statement that occurs in the observed samples has the possibility of being included, 2) the order and the control-flow structure of the statements in the execution trace reflect that of the observed samples and 3) the execution trace is likely to contain program statements and structure from most of the observed samples. We have shown that MCETs offer a more effective solution to the stochastic simulation - dynamic program slicing problem when evaluated through expected and cumulative relevance. While the MCET solution is not as efficient as existing approaches its inefficiency is bounded, it keeps expected user time at acceptable levels, and we conclude its significant improvement in effectiveness in the general case over more efficient solutions is an acceptable tradeoff for the extra computation time and I/O required.

MCETs are a new research artifact. Thus, their utility in analyzing behaviors of stochastic simulations is an ongoing investigation. However, based on their effectiveness in our solution to the stochastic simulation - dynamic program slicing problem we expect that they will significantly improve informed analysis of stochastic simulations. In future work, we expect to demonstrate how MCETs can be employed to improve software testing methods and measures for statement and path coverage of stochastic simulations. Also, we will explore how fault localization tools, currently only applicable to deterministic programs, can be adapted to stochastic simulations by employing representative execution traces.

ACKNOWLEDGMENTS

We gratefully acknowledge support from our colleagues at the University of Virginia and funding from Science Applications International Corporation (SAIC).

REFERENCES

Allen, A. O. 1990. *Probability, statistics, and queueing theory with computer science applications*. San Diego: Academic Press Professional, Inc.

Arthur, W. 1999. Complexity and the Economy. *Science* 284:107-109.

Baeta-Yates, R., and B. Ribeiro-Neto. 1999. *Modern Information Retrieval*. New York: Addison-Wesley.

Berg, A. 2004. *Markov Chain Monte Carlo Simulations and Their Statistical Analysis: With Web-based Fortran Code*. World Scientific Press: London.

Binkley, D., and K. Gallagher. 1996. Program Slicing. *Advances in Computers*, Volume 43, ed. Marvin Zelkowitz. Academic Press San Diego, CA.

Ducasse, M. 1999. A pragmatic survey of automated debugging. *Lecture Notes in Computer Science* 749:1-15.

Dunham, J. 2005. An Agent-Based Spatially Explicit Epidemiological Model in MASON. *Journal of Artificial Societies and Social Simulation* 9:(1).

Eclipse Foundation 2009. Eclipse Debugger. Available via <http://www.eclipse.org/> [accessed April 6, 2010].

Elderd, B., V. Dukic and G. Dwyer. 2006. Uncertainty in predictions of disease spread and public health responses to bioterrorism and emerging diseases. *Proceedings of the National Academy of Sciences* 103(42): 15693-15697.

Fu, M. 2002. Optimization for simulation: theory vs. practice. *Journal on Computing* 14(3): 192-215.

Gore, R 2010. Improved Methods and Measures for Computing Dynamic Program Slices in Stochastic Simulations: Efficiency Evaluation. Available via <http://www.cs.virginia.edu/~rjg7v/mcet/> [accessed April 10, 2009].

Glynn, P.W. 1987. Likelihood ratio gradient estimation: an overview. In *Proceedings of the 1987 Winter Simulation Conference,* ed. A. Thesen, H. Grant and W. David Kelton, 366–374. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Hartman, C. 1996. *The Virtual Muse: Experiments in Computer Poetry*. Hanover: Wesleyan University Press.

Heidelberger, P. 1995. Fast simulation of rare events in queueing and reliability models. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 5(1):43-85.

Java 2010. Java Debugging Interface: Overview. Available via <http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jdi/index.html> [accessed April 6, 2010].

Jayaraman, G., V. P. Ranganath and J. Hatcliff. 2005. Kaveri: Delivering the Indus Java program slicer to Eclipse. In *Proceedings of Fundamental Approaches to Software Engineering Conference* 269–272.

Jones, J., and M. Harrold. 2005. Empirical Evaluation of the tarantula automatic fault localization technique. In *Proceedings of the 20th International Conference on Automated Software Engineering* 273-282.

Kenner, H., and O. Joseph. 1984. A Travesty Generator for Micros. *BYTE* 9 (12): 129–131.

Korel, B., and J. Rilling. 1997. Application of dynamic slicing in program debugging. In *Proceedings of the 3rd International Workshop on Automated Debugging*, ed. M. Kamkar, 43-57. Linköping: Linköping University Electronic Press.

Law, A. M., and W. D. Kelton. 2000. *Simulation modeling & analysis.* 3rd ed. New York: McGraw-Hill, Inc.

Meyn, S. P., and R.L. Tweedie. 2009. *Markov chains and stochastic stability*. 2nd ed. London: Springer-Verlag.

Ólafsson, S., and L. Shi. 1999. Optimization via adaptive sampling and regenerative simulation. In *Proceedings of the 1999 Winter Simulation Conference,* ed. P. A. Farrington, H. B. Nembhard, D. T. Sturrock and G.W. Evans, 666-672. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Phet 2010. Phet: Self-Driven Particle Simulation. Available via <http://phet.colorado.edu/simulations/sims.php?sim=SelfDriven_Particle_Model> [accessed March 3, 2010].

Tip, F. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3(3):121-189.

Wang, T., and A. Roychoudhury. 2008. Dynamic slicing on java bytecode traces. *ACM Transactions on Programming Languages and Systems* 30(2):1–49.

Weiser, M. 1984. Program Slicing. *IEEE Transactions on Software Engineering* 10(4):352-357.

Whipple, C. 1996. Can nuclear waste be stored safely at yucca mountain? *Scientific America* 274(6):72-79.

Whittaker, J. and M. Thomason. 1994. A Markov Chain Model for Statistical Software Testing. *IEEE Transactions on Software Engineering* 20(10):812-824.

AUTHOR BIOGRAPHIES

ROSS GORE is a Ph.D. Candidate in Computer Science at the University of Virginia. His email address is <rjg7v@virginia.edu>.

PAUL F. REYNOLDS, JR. is a Professor of Computer Science at the University of Virginia. He has conducted research in Modeling and Simulation for over 30 years, and has published on a variety of topics including parallel and distributed simulation, multi-resolution modeling and simulation. His email address is <reynolds@virginia.edu>.