

Understanding Unexpected Behaviors in Exploratory Simulations

Ross Gore
October 15, 2009

School of Engineering and Applied Science
University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740

A Dissertation Proposal
Presented in Partial Fulfillment
of the Requirement for the Doctor of Philosophy Degree
at the University of Virginia
School of Engineering and Applied Science
Department of Computer Science

ABSTRACT

Simulations and computational models have become the common tool of subject matter experts (SMEs) in a variety of disciplines to explore systems with inherent uncertainty. Predictions from these models and simulations have entered the mainstream of critical public policy and research decision-making practices. Unfortunately, methods for gaining insight into unexpected simulation outcomes have not kept pace. SMEs need to understand and explain unexpected behaviors in exploratory simulations to determine if the behaviors reflect an error or if they represent new knowledge in the discipline. Common practice is to apply classic debugging techniques to identify the program statements and interactions that lead to the unexpected behaviors. This practice is largely manual, it can consume years of effort, and it will not scale as models increase in complexity. Automation of at least a portion of the process has become essential. The automated process proposed here, CPS, will combine program slicing and causal networks in a novel manner to identify potential causality between program statements and unexpected behaviors. CPS will facilitate focusing SME attention on understanding and explaining the interactions of program statements whose execution causes variable state changes that are most relevant to the cause of unexpected behaviors. Issues that make the proposed approach research challenging include: identifying prior knowledge in exploratory simulation that can be employed by causal networks, efficiently sampling variable states and dynamic program slices and identifying an approach to cluster similar dynamic program slices. Evaluation of CPS will employ established methods for evaluating human productivity tools and information retrieval metrics. The effectiveness of CPS will be compared to that of the leading debugging, program comprehension and fault localization tools. The thesis of the proposed work is that by these measures CPS will be deemed more effective than existing tools and will be a useful contribution by bringing automation to a challenging task.

1. INTRODUCTION

The research proposed here addresses improved methods and tools for explaining unexpected outcomes in simulations used for exploratory analysis. The combination of existing software analysis methods such as program slicing, and data analysis methods such as causal networks, presents an opportunity for a beneficial outcome. Among the expected benefits are: incorporation of a higher degree of automation in a process that is currently largely labor intensive, development of a more effective methodology for the process of understanding unexpected outcomes, and development of a more effective tool for retrieving program statements relevant to unexpected outcomes.

This proposed research is needed. Simulation and computational modeling have become mainstays for exploration in a broad landscape of disciplines. For example, subject matter experts (SMEs) in biology, chemistry, physics and astronomy employ models and simulations of *systems* (Fortnow 2009), where a *system* is defined as the domain or process of interest to the SME. The National Science Foundation expects that the use of models and simulations in more disciplines will increase over time (NSF 2007). Uncertainty about system structure is inherent in the modeling and simulation of many systems. This uncertainty results in unexpected simulation outcomes. As the use of models and simulations increases, the need for an automated methodology for facilitating SME understanding and explanation of the unexpected outcomes is increasing. Further compelling arguments regarding the need and importance of this proposed research follow the presentation of relevant, currently accepted definitions.

The combination of models, simulations, uncertainty and exploration possesses an established terminology and structure that provides a framework for the proposed research. SMEs develop models by specifying logical and mathematical relationships to describe how a system works (Law and Kelton 2000). The computational evaluation of the model is a simulation of the system the model represents, and the simulation is the artifact that results from modeling systems (Law and Kelton 2000). SMEs construct two different types of models for systems: *consolidative models* and *exploratory models* (Bankes 1993). A *consolidative model* is the description of the logical and mathematical relationships within the system, where the relationships are accepted by SMEs in the field. The computational evaluation of a consolidative model is a *consolidative simulation*. The simulation consolidates the information about the system into a particularly useful form and can be used as a surrogate for the system itself (Bankes 1993). However, for other systems there exists uncertainty about some of the logical and mathematical relationships. The uncertainty about these relationships exists because experiments cannot be realized due to constraints imposed by the nature of these systems (Whipple 1996; Arthur 1999; Hooke 2000; Cha 2005; Elder 2006; NSF 2006). As a result consolidative models cannot be constructed for these systems, instead SMEs form *exploratory models* by making estimations about the uncertain mathematical and logical relationships within the system (Bankes 1993). Constructing an exploratory model is a theory construction task where the model is the expression of the theory, and the simulation is the research artifact that allows the theory to be evaluated. Other simulations of the same system, any data from experiments, or other SME opinions in the field are used to test the theory. This gives exploratory simulations their experimental nature (Trenouth 1991).

Model requirements exist for both consolidative and exploratory models to identify and document, in native language, required model output values for model input values. These requirements are generally accepted in the field and established through experiments and observation of the system (Law and Kelton 2000; Ghezzi et al. 2003). Simulation outputs that are not defined by the model requirements and do not match simulations of the same system, data from experiments, or SME opinions are *unexpected behaviors*. Unexpected behaviors in consolidative simulations are rare because of the completeness of consolidative model requirements and the accepted SME knowledge in the field. However, due to the incomplete nature of exploratory model requirements and uncertainties regarding relationships within the system, unexpected behaviors are common in exploratory simulations. SMEs require the capability to understand and explain unexpected behaviors in exploratory simulations to determine if the behavior is due to an error or if it represents new knowledge about the system. If the behavior does represent new knowledge the model requirements are modified to reflect this established knowledge for the system. The process of understanding and explaining unexpected behaviors is further complicated in exploratory simulations because of the use of

continuous and discrete random variables to estimate the uncertain relationships in the system. The use of random variables in an exploratory simulation results in outputs that are random (Law and Kelton 2000). As a result exploratory simulations often exhibit unexpected behaviors that are not immediately repeatable.

An effective methodology for facilitating SME understanding and explanation of unexpected behaviors in exploratory simulations can affect millions of people and billions of dollars. Exploratory simulations are commissioned and employed by decision makers for policy analysis to answer questions where experimentation and observation is infeasible. For example, the inability of researchers to explain the results of the simulation, Episims, has led to public policy debate. Episims models the nationwide spread of the smallpox virus under various vaccination strategies (Eubank et al. 2004). Previous simulations of the smallpox virus show that a targeted vaccination strategy, where individuals most likely to spread the disease are targeted for vaccination, manages disease spread as well as a mass vaccination for the entire population. However, the results of Episims show that in the event of a smallpox outbreak the disease spread under the targeted vaccination strategy is much more severe than under the mass vaccination strategy. The difference between these predictions has led to policy debate over "whether or not it's necessary to synthesize enough smallpox vaccine for the entire country" (Cha 2005). The Institute of Medicine of the National Academies has published a collection of critical opinions of the predictions from Episims. The chief complaint is that the simulation developers cannot provide a clear explanation for the difference between their predictions under these vaccination strategies and previously established estimates (Baciu et al. 2005). Methodology to facilitate the understanding and explanation of Episims' behaviors would help to resolve this debate.

To address the need for explaining unexpected behaviors in exploratory simulations, I propose Causal Program Slicing (CPS), which combines program slicing and causal networks to provide an effective tool for SMEs to explore unexpected behaviors in exploratory simulations. Program slicing is used in a variety of software analysis tools to extract statements that are relevant to a particular computation within the program. Causal networks are used to identify the causal structure of measurable variables in medical, economic and social science studies. The combination of causal networks with program slicing is attractive because causal network analysis can be focused on identifying how variables interact to cause a phenomenon. This emphasis on discovery does not require an unexpected behavior in an exploratory simulation to be classified as correct or an error before causal networks can be applied. Furthermore, because causal networks are based in probability theory they are applicable to exploratory simulations which use random variables. These tools motivate my thesis statement:

The purposeful combination of program slicing and causal networks will result in a more effective methodology than existing tools and practices for facilitating SME understanding and explanation of unexpected behaviors in exploratory simulations.

CPS will demonstrate this thesis statement. CPS will employ program slicing and causal networks in combination to identify how the program statements of an exploratory simulation change the states of variables within the simulation to cause an unexpected behavior. Furthermore, CPS will focus SME attention on understanding and explaining the interactions of program statements whose execution causes variable state changes that are most relevant to the cause of the unexpected behavior. The goal of CPS is to enable users to gather insight into unexpected behaviors of exploratory simulation by understanding and explaining the interactions of the program statements that are most relevant to the cause of the unexpected behavior.

Two groups of SMEs employ exploratory simulations. The first is *source code level* SMEs who implement exploratory simulations in source code written in a high-level programming language. Accordingly, these SMEs are comfortable receiving insight into unexpected behaviors of exploratory simulations in terms of source code written in a high-level programming language. The second group of SMEs is *graphical tool or library* SMEs who implement exploratory simulations using graphical tools and libraries such as Simulink (Mathworks 1999), Java Modeling Tools (Bertoli et. al 2007), and VisSim (Planung Transport Verke 2005). These SMEs need insight into unexpected behaviors in exploratory simulations expressed in the same graphical tool or library in which the simulation is designed and implemented. CPS will be designed, implemented and evaluated for source code level SMEs. However, any decisions that could preclude CPS from being adapted for use by graphical tools or library SMEs will be

weighed carefully. The effectiveness of CPS will be evaluated for the source code level SMEs through quantitative and qualitative evaluation criteria, as discussed further in Section 4.

2. BACKGROUND AND RELATED WORK

CPS draws on the areas of program slicing, causal networks and cluster analysis and it is related to work in the program understanding and fault localization communities. A review of this work follows.

2.1 PROGRAM SLICING

Program slicing is a decomposition technique that extracts statements relevant to a particular computation within the program (Weiser 1981). A program slice provides the answer to the question, "What program statements affect the computation of variable v at statement s ?" (Binkley and Gallagher 1996) 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.

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

Figure 1: (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.

CPS uses static program slicing to identify those program statements which may affect an unexpected behavior in an exploratory simulation. This limits, sometimes significantly, the number of program statements

and variable states that need to be considered in the CPS analysis (Atkinson and Griswold 2001). CPS also collects dynamic program slices to quantify and analyze the different executions that are possible for an exploratory simulation given a set of input configurations.

2.2 CAUSAL NETWORKS

A Bayesian network is a graphical model that compactly represents the joint probability distribution of a set of variables in a directed acyclic graph (DAG). Formally, a Bayesian network is a pair (G, Θ) , where: G is a DAG over the set of variables Z , and Θ is a set of Conditional Probability Tables (CPTs), such that a CPT exists for each variable $z \in Z$. A Bayesian network is formed by the user supplying any prior knowledge about the independence relationships among the variables in Z . The user also supplies any *temporal information* that s/he possesses about the relationship among variables. Temporal information about two variables p and q means that a change in variable p occurs in time before a change in variable q . The user supplied temporal information is used to orient edges in the DAG, G . The prior knowledge is then augmented with a CPT for each variable in Z . Given a variable $z \in Z$, the CPT for z contains an entry for each value of z , for all values of the other variables $x \in Z$. This allows the Bayesian network to provide the probability of x conditional on z , $P(x|z)$, for all values of z for every instantiation of x (Pearl 1989; Darwiche 2009).

Using the prior knowledge supplied by the user and the CPTs for each variable, existing inference algorithms can be employed to construct and orient the edges between vertices forming the DAG, G , of the Bayesian network. In the DAG, each vertex represents a variable $z \in Z$ and each edge represents a lack of conditional independence between the variables. Within the network the following condition, called the Markov Condition must hold, “each variable is conditionally independent of its non-descendants given its parent variables.” The following notation is introduced to formally describe the Markov Condition. Given a variable, z in a DAG, G :

- Parents(z) is the set of variables N with an edge from N to z .
- Descendants(z) is the set of variables M with a directed path from z to M .
- Non-Descendants(z) are all variables in DAG, G other than z , Parents(z) and Descendants(z).
- $a \perp\!\!\!\perp B \mid C$ denotes that the variable a is independent of the set of variables B conditional on the set of variables C .

Given this notation the DAG, G of the Bayesian network compactly represents the following independence statements: $z \perp\!\!\!\perp \text{Non-descendants}(z) \mid \text{Parents}(z)$ for all $z \in Z$ (Pearl 1989; Darwiche 2009).

Bayesian networks allow users to graphically model the dependencies among variables but do not allow users to graphically model causality; causal networks address this shortcoming. A causal network is a Bayesian network, where the user is willing to make the assumption that the edges imposed from the Markov Condition denote causality. This assumption is called the Casual Markov Assumption: “Given a variable z , Parents(z) denotes the direct causes of z and Descendants(z) denote z ’s effects.” The assumption is that a lack of conditional independence between variables signifies a causal relationship (Spirtes et al. 2000).

The goal of CPS is to employ causal networks to represent the interactions of variable states causing an unexpected behavior in an exploratory simulation. The causal networks will be used by SMEs to understand the interactions within an exploratory simulation and explain why an unexpected behavior arises.

2.3 PROGRAM UNDERSTANDING

Understanding how a program feature is implemented is a major research area of program understanding, especially when maintaining or modifying an existing program. Gallagher and Lyle (Gallagher 1989; Gallagher and Lyle 1991) use static slicing for the decomposition of a program into a set of components capturing part of the original program’s behavior. Several other researchers have used dynamic slices to identify code in legacy systems that relate to a specified feature (Kang and Bierman 1996; Lakhota and Deprez 1998; Wong et al. 2000; Wilde et al. 2001). Eisenbarth (Eisenbarth et al. 2001; Eisenbarth et al. 2003) has incorporated static and dynamic slices along with concept analysis to further automate the same process.

Ernst's program understanding work, Daikon, is an approach that is closely related to CPS. The approach offers an alternative to requiring programmers to annotate code with invariants. Daikon executes a program on a collection of inputs and extracts variable values and then infers the invariants (Ernst et al. 2006). The invariants assist users in understanding and maintaining software. Recently, Brun and Ernst have extended the Daikon approach to finding latent code errors via machine learning over program executions (Brun and Ernst 2004). The technique generates machine learning models of program properties known to result from errors, and applies these models to program properties of user-written code to classify and rank properties that may lead to errors. Given a set of properties produced by the program analysis, the technique selects a subset of properties that are most likely to reveal an error.

2.4 FAULT LOCALIZATION

Work on automatic fault location closely matches CPS' proposed goal of understanding and explaining the causes of unexpected behaviors in exploratory simulations. The goal of fault localization is to identify a fault or bug created by a programmer in source code. The fault creates an infection in the program state during execution, causing an externally observable error. Once the error is observed fault localization tools are applied (Zeller 2002). An initial solution to fault localization records the dynamic program slice from an execution that passes a test case and the dynamic program slice from an execution that fails the same test case. Program statements within the dynamic program slice failing the test case but not in the slice passing the test case are presented to the user (Agrawal et al. 1993).

Jones et al. significantly improved fault localization solutions by offering a statistic to determine the likelihood that a program statement in a slice that fails a test case contains a fault (Jones et al. 2002). Reinieris offered a nearest neighbor algorithm to analyze the distance between statements in passing and failing executions to attempt to compute the same statistic more accurately (Reinieris and Reiss 2003). Delta Debugging is a different approach to fault localization that isolates the causes of failing test cases by assessing outcomes of altered executions of the program to determine changes in the program state that create the difference in the test case outcomes (Zeller 2002; Cleve and Zeller 2005). However, all fault localization tools require that a user distinguish between valid and invalid behavior (an error). This is not possible for unexpected behaviors in exploratory simulations. Furthermore, these tools are not applicable to software that uses random variables.

2.5 CLUSTER ANALYSIS

Clustering algorithms are often employed to enable SMEs to understand phenomena of interest in a variety of fields. The goal of clustering is to separate a finite unlabeled data set into a finite and discrete set of clusters (Xu and Wunsch 2005). The data within clusters is grouped according to whether a piece of data is *similar* to the other pieces of data in the cluster. Clustering is ubiquitous and a wealth of clustering algorithms and definitions of similarity have been developed to solve different problems in specific fields. However, there is no clustering algorithm or definition of similarity that can be universally employed to solve all problems (Xu and Wunsch 2005). Despite the lack of a universal clustering algorithm, several different research efforts have successfully applied clustering algorithms to group program executions (Liu et al. 2007; Leon et al. 2007). CPS will use clustering analysis to group similar executions of exploratory simulations together. Then, CPS will generate a causal network that represents how interactions of the variable states and program statements within the cluster cause the unexpected behavior. The expectation is that many different executions fall into a few groups, that similar executions share a similar influence on the unexpected behavior in an exploratory simulation, and that clustering analysis can group these executions together.

3. PROPOSED RESEARCH

Four research objectives in support of realizing CPS are presented. These will demonstrate my thesis and constitute novel work in computer science:

1. *The demonstration that static analysis tools can be used to identify prior knowledge about independence and temporal relationships among variable states within exploratory simulations.*
2. *The demonstration that the variable states and executable dynamic program slices within exploratory simulations can be efficiently sampled.*
3. *The demonstration that causal networks revealing the structure of program statements causing an unexpected behavior can be generated for clusters of exploratory simulation executions using the data provided in research objectives (1) and (2).*
4. *The demonstration that the clusters of causal networks provided in research objective (3) are more effective in facilitating user understanding and explanation of the unexpected simulation behaviors in exploratory simulations than existing tools and practices.*

In the remainder of this section the proposed shape of CPS, a research plan to complete the development of CPS and the expected contributions of CPS are presented.

3.1 THE SHAPE OF CPS

The goal of CPS is to employ causal networks to represent the interactions of variable states causing an unexpected behavior in an exploratory simulation. CPS begins with the SME identifying program state representing the unexpected behavior. Once the unexpected behavior has been identified, the CPS process will start with the application of existing static analysis tools to the exploratory simulation exhibiting unexpected behavior to provide prior knowledge about the independence and temporal relationships among variable states. Next, the exploratory simulation will be instrumented to collect the values of each variable state which can affect the unexpected behavior. Then, the exploratory simulations will be run for a set of SME supplied inputs to collect the data required to generate the CPT (conditional probability table) associated with each variable state. The executions resulting from the set of exploratory simulation runs will be grouped based on their similarity using cluster analysis. Finally, existing algorithms will be employed to generate a causal network for each cluster of executions. The resulting causal networks will identify, for each cluster of executions, how the interactions of variable states cause the unexpected behavior. These causal networks will be used by the SME to understand and explain the unexpected behavior in the exploratory simulation.

```

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

```

Figure 2: A variation of the program in Figure 1 that uses random variables.

The program shown in Figure 2 will be used as an example throughout this section to motivate and describe each component of CPS. The program is a variation of the program shown in Figure 1(a) that uses random variables. The program takes an integer input n and computes the sum of n samples taken from a

uniform random number distribution between 0 and n, where each sample is an integer. If the sum of the n random samples is evenly divisible by n then -1 is assigned to x. If the sum of the n random samples is not evenly divisible by n then sum is assigned to x. The value of x is printed. Finally, the average of the n random samples is computed and printed.

3.1.1 APPLYING STATIC ANALYSIS TO PROVIDE PRIOR KNOWLEDGE FOR CPS ANALYSIS

CPS will begin with the user identification of the state(s) of the program that represents the unexpected behavior. The program statement in the source code of the exploratory simulation at which this state can be observed is identified by its line number, *s*. The variable storing the value of interest related to the unexpected behavior is identified by the variable, *v*. In the example in Figure 2, the unexpected behavior is the printed value of x in statement 13. This behavior is identified with {13, x}.

Next, static program slicing will be applied using the static slicing criterion {*s*, *v*}. The static program slice will identify all statements in the simulation's source code containing variables that may influence the unexpected behavior. Figure 3 shows the static program slice of the program in Figure 2 given the static slicing criterion representing the unexpected behavior, {13, x}. Notice statements 5, 14 and 15 which compute the average of the n randomly drawn integers and are not relevant to the computation of x in statement 13 and thus have been removed. In preliminary work, static program slicing has been used successfully within CPS to conservatively focus the analysis on the variable states and program statements that affect the unexpected behavior within the exploratory simulation (Gore and Reynolds 2009a; Gore and Reynolds 2009b).

```
1   read(n);
2   i := 1;
3   x := 0;
4   sum := 0;

6   while i<= n
7       sum := sum + rand (0, n);
8       i := i + 1;
9   end
10  if (sum mod n == 0)
11      x := -1;
12  else
13      x := sum;
14      print (x);
15  end
```

Figure 3: A static slice of the program using criterion {13, x}.

Next, static data dependence and flow analyses will be applied to the statements in the static program slice. The analyses will be used to provide prior knowledge about the independence and temporal relationships among variable states in the slice. This application of these analyses is proposed work. The expectation is that static data dependence analysis will identify independence relationships among variable states and static data flow analysis will identify the temporal relationships among variables. The challenges associated with this work and directions for solution are discussed further in Section 3.2.1.

By employing these static analysis techniques the scope of the CPS analysis will be conservatively bounded affording more efficient, yet still complete analysis. Furthermore, the prior knowledge about the independence and temporal relationships of variable states will be supplied to CPS to help generate the causal network of the interactions causing the unexpected behavior without any effort from the SME.

3.1.2 PREPROCESSING EXPLORATORY SIMULATION PROGRAM STATEMENTS

Following the application of static data dependence and flow analyses the statements in the static program slice will be preprocessed by a CPS preprocessor. The preprocessor will instrument the source code in the exploratory simulation to collect the value and address of each variable state in the static program slice when

the exploratory simulation is executed. These variable state value samples will be used to form the CPTs required for the causal network. Currently, the CPS preprocessor does not capture the values of all array component variable states and all variable states within loops (Gore and Reynolds 2009a). It also does not capture the addresses of the program statements in which these variable states appear. Instead, the preprocessor aggregates all components of an array in a single variable state sample and all iterations of a loop into a single variable state sample. Refining the CPS preprocessor in these capacities is future work. The challenges associated with this future work and directions for solution are discussed further in Section 3.2.2.

3.1.3 SAMPLING THE VARIABLE STATES AND DYNAMIC PROGRAM SLICES

Once the preprocessing step is complete, the SME will identify the set of input parameter configurations for which s/he is interested in gaining insight into the unexpected behavior. The exploratory simulation will then be executed for each configuration. The source code inserted by the CPS preprocessor will collect the samples for the value of each variable state and the address of each executed program statement which is included in the static program slice. The collection of addresses for each executed program statement will form a dynamic program slice. Each dynamic program slice will be associated with the set of samples that it generated.

Recall, for exploratory simulations which use random variables, there exist different possible outputs (or behaviors) for a fixed input (Law and Kelton 2000). Most software analysis tools for sequential programs assume that for each input there exists only one possible output and that for each input there exists only one possible dynamic program slice to be executed. The first assumption is never true for exploratory simulations that employ random variables and often the second assumption is not true either.

Figures 4(a) and 4(b) help elucidate the issues that can arise in analysis when these assumptions are made. Figures 4(a) and 4(b) show the two possible dynamic program slices using criterion $\{n = 5, 13, x\}$ for the program in Figure 2. Figure 4(a) shows the dynamic program slice, when the sum of n random numbers is not evenly divisible by n . Figure 4(b) shows the slice when the sum is evenly divisible by n .

<pre> 1 read(n); 2 i := 1; 3 x := 0; 4 sum := 0; 5 average := 0; 6 while i<= n 7 sum := sum + rand (0, n); 8 i := i + 1; 9 end // (sum mod n == 0) == false 12 x := sum; 13 print (x); 14 average := sum/n; 15 print (average); </pre>	<pre> 1 read(n); 2 i := 1; 3 x := 0; 4 sum := 0; 5 average := 0; 6 while i<= n 7 sum := sum + rand (0, n); 8 i := i + 1; 9 end // (sum mod n == 0) == true 11 x := -1; 13 print (x); 14 average := sum/n; 15 print (average); </pre>
(a)	(b)

Figure 4: (a) One possible dynamic program slice of Figure 2 using criterion $\{n=5, 13, x\}$. (b) Another possible dynamic slice of Figure 2 using criterion $\{n = 5, 13, x\}$.

Analysis that only employs one dynamic program slice for the program in Figure 2 will not capture all the possible behaviors. Furthermore, even if only one of these dynamic program slices is possible different values for a single variable state are possible. In the program in Figure 2, given input $n=5$, the value of the variable state of x in statement 12 can be 0, 5, 10, 15, 20, or 25 if the dynamic program slice in Figure 4(b) is executed. Currently, CPS addresses these issues by performing Monte-Carlo sampling for each input configuration multiple times when the exploratory simulation exhibiting unexpected behavior uses random variables (Gore and Reynolds 2009b; Gore and Reynolds 2009c). However, this solution to sampling is inefficient, in terms of time and space. Developing a more efficient approach to sampling variable states and

dynamic program slices of exploratory simulations is future work. This challenge along with several directions for solutions is described in Section 3.2.3.

3.1.4 APPLYING CLUSTER ANALYSIS TO GROUP THE SAMPLES

Following the CPS collection of a representative sample of dynamic program slices and variable states for the SME supplied set of input configurations for the exploratory simulation, cluster analysis will be applied to group similar dynamic program slices together. The application of cluster analysis addresses the following problem: many different dynamic program slices can be possible for a set of SME supplied input configurations. Each configuration can result in at least one different dynamic program slice. When the exploratory simulation uses random variables many dynamic program slices may be possible for each configuration. For some exploratory simulations over 10,000 different dynamic program slices are possible for a single SME supplied input configuration (Gore and Reynolds 2009c).

The current solution employed by CPS to address this problem is to generate a single causal network for all the dynamic program slices collected for the set of SME supplied input configurations (Gore and Reynolds 2009a; Gore and Reynolds 2009b). The example program in Figure 2 and a SME supplied configuration set of only $n=5$ elucidate the shortcoming of this solution. Recall, for this example, the unexpected behavior is the value of x in statement 13. The single causal network generated for this example is shown in Figure 5(a). The causal network in Figure 5(a) aggregates the dynamic program slice and the associated variable state values shown in Figure 4(a) with the dynamic program slice and the associated variable state values shown in Figure 4(b). This is the shortcoming of the single causal network solution; the SME cannot determine the different interactions of the program statements which cause the program state representing the unexpected behavior to have different values because the different dynamic program slices are aggregated.

In this example the dynamic program slices should be grouped into two clusters: one cluster that corresponds to the dynamic program slice in Figure 4(a) and a second cluster that corresponds to the dynamic program slice in Figure 4(b). The causal networks generated for each of these clusters are shown in 5(b) and 5(c) respectively. Here, the different interactions causing the program state representing unexpected behavior to have different values are evident to the SME. Identifying an existing clustering algorithm and developing a definition of similarity for exploratory simulation dynamic program slices that will enable clustering to be effectively applied within CPS is future work. This challenge along with several directions for solutions is described in Section 3.2.4.

3.1.5 GENERATING CAUSAL NETWORKS OF THE PROGRAM STATEMENTS

Once the cluster analysis within CPS has grouped the different dynamic program slices, a causal network will be generated for each cluster. Each network will be generated from the prior knowledge identified by the static analysis tools and the samples of the variable states associated with the dynamic program slices in the cluster. CPTs will be formed for each unique variable state address from the samples associated with the dynamic program slices forming the cluster. The variable states will serve as the variables in the network.

Next, established algorithms will be employed to construct causal networks describing the interactions among the variable states which cause the unexpected behavior. The result will be a chain of variable states describing how each variable state influences the other variables, and the unexpected behavior. The proposed combination of causal networks with program slicing enables the influence variable states have on one another and the unexpected behavior to be quantified.

The strength of a causal influence is measured as the absolute value of the correlation coefficient (between $[0, 1]$ inclusive) between two variable states; given that the two variable states cannot be made conditionally independent by conditioning on any other variable or set of variables within the causal network. This condition is reflected through an edge between the two variables in the network (Spirtes et al. 2000).

Each variable state in the causal network with a causal influence that is over a SME specified threshold will be mapped back to the program statement that caused the variable's state to change. Finally, a graph of the chain of program statements that have a causal influence on the unexpected behavior will be displayed to

the SME. The graph is annotated with the causal influence each program statement has on the unexpected behavior or another program statement over threshold. The graph focuses SME attention on understanding those statements in the program’s source code with the strongest causal influence on the unexpected behavior.

The notion of causal influence does not directly translate from the open world where causal networks are often applied to the closed world of exploratory simulation. The quantified causal influence employed by CPS will not have semantic meaning in terms of cause and effect between variable states and program statements. Instead, causal influence will be used in a manner similar to the page ranking heuristics used by Google's page rank algorithms (Langville and Meyer 2006). This idea, that the interactions of some variable states are more important to the explanation of a behavior than others, is established within the fault localization community (Weiser 1981; Zeller 2002; Cleve and Zeller 2005). CPS has successfully generated causal networks for dynamic program slices from exploratory simulations in a variety of fields (Gore and Reynolds 2008; Gore and Reynolds 2009a; Gore and Reynolds 2009b). However, exploring other quantitative definitions of causal influence to improve CPS’ effectiveness in facilitating SME understanding and explanation of unexpected behaviors is future work. This exploration is described in Section 3.2.5.

CPS will also specify to the SME how frequently each of the different causal networks associated with the clusters is executed for the set of SME supplied input configurations. This allows the SME to understand how often a given causal network reflects the interactions of the program statements causing the unexpected behavior. The two causal networks associated with the two clusters formed from applying CPS to the example program in Figure 2 are shown in Figure 5(b) and 5(c). The frequency of each cluster of dynamic program slices is also provided. It is important to note that even in such a small example two different clusters are needed and the causal network of each cluster has a very different structure and frequency of execution.

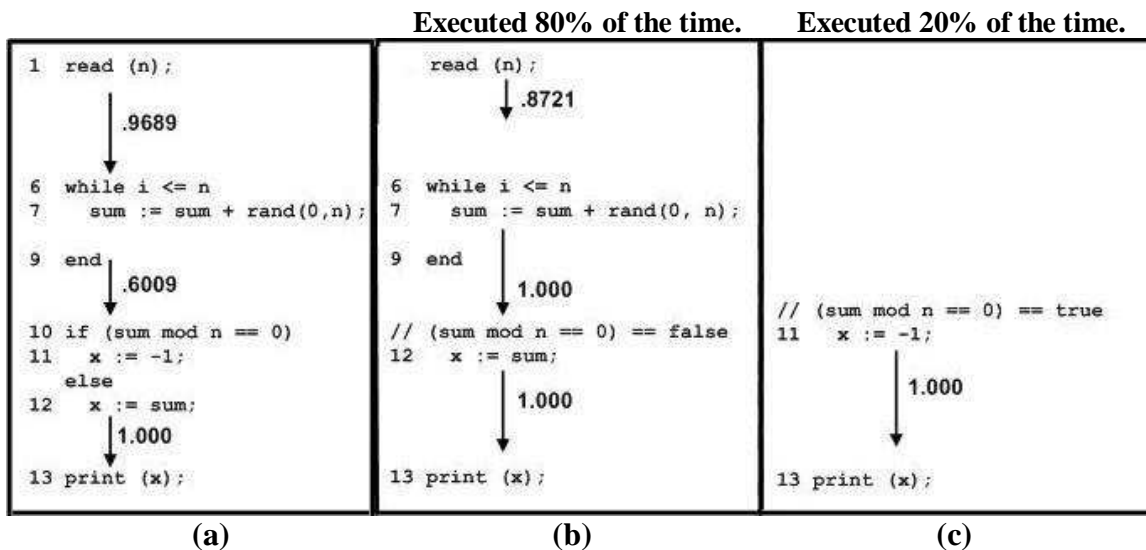


Figure 5: (a) A single causal network for all the dynamic program slices in Figure 2. (b) The causal network for the dynamic program slices in Figure 4(a). (c) The causal network for the dynamic program slices in Figure 4(b). All three causal networks (a-c) have a SME specified causal influence threshold of .5.

3.2 RESEARCH PLAN

The research plan for this proposed work consists of the sequence of tasks that must be completed in order to meet the research objectives presented at the beginning of this section.

1. *Develop an approach to applying static analysis tools to identify prior knowledge about temporal and independence relationships among variable states within exploratory simulations.* This prior knowledge must be in a form that can be used by causal networks. **Challenges:** Causal networks have not previously been applied to software. As a result, identifying how prior knowledge about variable states within an

exploratory simulation can be supplied to causal networks is a new research challenge. Two types of prior knowledge need to be supplied to the causal network: information regarding the independence relationships among variable states and the temporal relationships among variables states.

Static data dependence analysis is expected to provide prior knowledge about the independence relationships among variables states. The analysis identifies any variable state that may reach a candidate variable reference (Ranganath 2006). The expectation is that for each candidate variable state in each remaining statement CPS will apply static data dependence analysis tools to determine the variable states in remaining statements that are data dependent on the candidate variable state.

Static data flow analysis is expected to provide prior knowledge about the temporal relationships among variable states. Static data-flow analysis is a technique for gathering information about the possible set of values calculated within various variable states for a given program variable (Ranganath 2006; Chang and Jo 2002). Data-flow analyses often employ a control flow graph (CFG) for the program which is used to determine the variable states where a particular value assigned to a variable might propagate. The expectation is that CPS will apply static data flow analysis tools to identify the temporal order of the variable states which propagate values to the unexpected behavior.

2. ***Refine the CPS preprocessor to capture the values of array component variable states and variable states within loops. Also capture the addresses of the program statements in which these variable states appear. Challenges:*** Due to the current implementation of CPS, which only generates a single causal network for the set of SME supplied input configurations, the CPS preprocessor does not collect these variable states and addresses. The future application of cluster analysis in CPS is expected to alleviate this restriction. A refined CPS preprocessor will use a program tracing utility, such a ptrace (Padala 2002a; Padala 2002b) to capture all variable states values and the addresses of all executed program statements that affect the program state(s) representing the unexpected behavior.
3. ***Develop an approach to efficiently sample variable states and dynamic program slices within an exploratory simulation; efficiency is measured in terms of space and time. Challenges:*** CPS requires a representative set of samples of variable states and dynamic program slices to generate causal networks of the interactions of variables states within an exploratory simulation. For exploratory simulations which employ random variables each input configuration needs to be sampled multiple times. Applying a straightforward approach of repeatedly executing the simulation and recording the variable states and dynamic program slice for each input configuration will require an overwhelming amount of space and time. Exploring symbolic execution algorithms used within the model checking community to improve the time efficiency of generating these samples is promising (Clarke et al. 2000). Also compression algorithms will be explored to improve the spatial efficiency of storing the dynamic program slice executions that result from an exploratory simulation (Wang and Roychoudhury 2004).
4. ***Develop a definition of similarity and identify an existing clustering algorithm to effectively group exploratory simulation dynamic program slices. Challenges:*** Actual exploratory simulation dynamic program slices will not be as simple to cluster as the dynamic program slices of the program in Figure 2. Identifying the characteristics of dynamic program slices that will be included in the definition of similarity required by clustering algorithms will be challenging. Furthermore, there is not a unified clustering framework that works for all problems. The expectation underlying the application of cluster analysis is that many of the dynamic program slices share similar characteristics, and those characteristics can be identified and categorized in a small number of groups. This expectation has been shown to be true of dynamic program slices within software employed in a variety of domains (Liu et al. 2007; Leon et al. 2007). The exploration for an effective clustering algorithm will begin with algorithms used in these research efforts.
5. ***Explore alternative quantitative definitions of causal influence to improve CPS' effectiveness in facilitating SME understanding and explanation of unexpected behaviors in exploratory simulations.***

Challenges: The absolute value of the correlation of a variable identified as a cause of another variable is used to measure causal influence in causal network community (Spirtes et al. 2000). However, it is not necessarily the definition of causal influence that will result in the most effective CPS analysis for SMEs tasked with understanding and explaining unexpected behaviors in exploratory simulations. Several different quantitative metrics have been developed in the fault localization community to identify program statements whose executions are most likely to cause a program run to fail a test case (Zeller 2002; Cleve and Zeller 2005; Renieris and Reiss 2003; Jones et al. 2002). Each of these established definitions will be explored to identify the definition of causal influence that offers the most effective CPS analysis for facilitating SME understanding and explanation of unexpected behaviors.

6. **Establish and conduct the evaluation of CPS. Challenges:** The details and challenges associated with the research setup and methodology of the CPS evaluation are discussed in Section 4.

3.3 EXPECTED CONTRIBUTIONS

1. **The demonstration that the information static analysis tools identify about independence relationships among variable states within exploratory simulations can be used effectively in causal networks as prior knowledge.** CPS will apply causal networks to facilitate SME understanding and explanation of unexpected behaviors in exploratory simulation. The demonstration that static analysis tools can provide the prior knowledge about variable states in software, required by causal networks, will enable other computer science researchers to explore applying causal networks to software for other purposes.
2. **The quantification of the probability distribution of dynamic program slices in exploratory simulations that employ random variables.** Currently, there is no software analysis methodology for programs that use random variables. This is due, in part, to the uncertainty associated with the dynamic program slice that is executed for a given input, for this class of software. CPS will address this issue enabling research in software analysis for programs using random variables.
3. **The development of a more effective tool for facilitating SME understanding and explanation of unexpected behaviors in exploratory simulations than currently exists.** The effectiveness of CPS for this contribution will be evaluated through qualitative evaluation criteria described in Section 4.
4. **The development of a more effective tool for retrieving program statements relevant to an unexpected behavior in an exploratory simulation than currently exists.** The effectiveness of CPS for this contribution will be evaluated through quantitative evaluation criteria described in Section 4.

4. EVALUATION

The success of CPS in addressing the thesis and objectives of the proposed research will be evaluated. The evaluation is described by measures of effectiveness (MOEs), measures of success (MOSs) and the existing tools, representing the state of the art, which CPS will be evaluated against.

The evaluation of CPS will require some research setup including: identifying existing case study exploratory simulations exhibiting unexpected behaviors and identifying SME collaborators. Previous efforts of the MaSTRI research group have established contacts in the departments of environmental science, materials science, mechanical and aerospace engineering, physics and neurological surgery at the University of Virginia. The use of these contacts as well as contacts within the department of computer science is expected to yield case study exploratory simulations exhibiting unexpected behaviors and SME collaborators that will benefit from CPS.

4.1 MEASURES OF EFFECTIVENESS

Two measures of effectiveness will be employed to evaluate CPS: Multi-dimensional In-depth Long-term Case Studies (MILCs) and the average precision measure used in the information retrieval community (IR) (Shneiderman and Plaisant 2006; Singhal 2001). The MILCs are employed to qualitatively measure how effective CPS is in facilitating SME explanation and understanding of unexpected behaviors in exploratory simulations. This is the goal of CPS. The average precision standard measures CPS' ability to retrieve program statements that are relevant to an unexpected behavior in an exploratory simulation. This is not the primary goal of CPS. However, successful evaluation of CPS with this measure will demonstrate quantitatively how CPS improves the state of the art in analysis of exploratory simulations. This state of the art capability, in part, will allow CPS to improve the state of the art in facilitating SME explanation and understanding of unexpected behaviors in exploratory simulations. If CPS is more effective than existing tools for the average precision measure it will not only show that the CPS methodology is successful in achieving its goal but that CPS also offers successful research directions in analysis of software that employs random variables.

1. ***CPS will be evaluated through five MILCs.*** MILCs define a process for studying the benefit of tools which enable user insight (Shneiderman and Plaisant 2006). The *multi-dimensional* aspect refers to using observations, interviews, surveys, and automated logging to assess the utility and effectiveness of a given tool to users. The *in-depth* aspect is the intense engagement of the tool developer(s) and domain expert user(s) to the point of developing a partnership. *Long-term* refers to studies that can be measured in 6-month terms. *Case studies* refers to the detailed reporting about a small number of individuals working on their own problems outside of the tool's development. There will be five different SME collaborators participating in the MILCs. In preliminary work CPS has been applied to simulations consisting of ~10K lines of source code. The size of the MILCs case study simulations will be within the same order of magnitude. The MILCs evaluation of CPS requires IRB approved surveys, interview questions and proposed observation sessions. The IRB is responsible for reviewing all university research for compliance with mandated research guidelines. Getting IRB approval requires iterative modification and can take several months. Thus, the drafting and submitting of surveys, interview questions and observation sessions must begin immediately to ensure that they will be available for evaluation.
2. ***CPS will be evaluated through the average precision measure.*** Average precision is an evaluation criterion established in the IR community. IR is the science of searching for documents, information within documents and relational databases (Singhal 2001). The problem of identifying the program statements that facilitate user understanding and explanation of unexpected behavior in exploratory simulations can be cast as an IR problem. Similarly, tools to isolate those program statements which are relevant to user understanding and explanation of the unexpected behavior can be considered IR tools. A program statement is relevant to an unexpected behavior if the execution of the program statement affects the value of the unexpected behavior (Tip 1995). Retrieving statements relevant to a specified behavior is the goal of static and dynamic program slicing tools (Tip 1995; Weiser 1981; Korel 1994).

In the context of retrieving program statements relevant to an unexpected behavior *precision* is defined as: the number of program statements retrieved by the tool that are relevant to the unexpected behavior divided by the number of program statements retrieved by the tool (Salton 1983; Singhal 2001). Average precision takes the precision statistic a step further by emphasizing returning statements that are more relevant to the unexpected behavior early to the user. Average precision is the average of the precision of each statement after truncating the list of statements returned by the tool at some number N.

Average precision is defined by the following: Average precision =
$$\frac{\sum_{r=1}^N P(r)}{\# \text{ of statements with relevance } > 0}$$

(Salton 1983; Singhal 2001). Here r is the rank assigned to a retrieved statement by the tool, N is the number of statements the tool must rank, and $P(r)$ is precision of the statement at a given rank. The list of

statements is returned to the user in ascending rank order. Each statement at a given rank has a precision between [0,1]. This precision is determined by how often the statement, when executed, affects the unexpected behavior. Within an exploratory simulation that does not use random variables, for a given specified input, the precision of a statement is either 0 or 1. The statement is either executed and affects the unexpected behavior for a specified input or it does not. However, for an exploratory simulation that does use random variables the precision of a statement may be between 0 or 1. In this case, the precision depends upon how often the random variables in the simulation cause a statement to be executed and affect the value of the unexpected behavior. For example, the precision of statement 11 in the program in Figure 2, given the behavior { 13, x } and the SME supplied input n=5, is .20 because statement 11 affects the computation of x in statement 13 20% of the time when the program is executed for the input n = 5.

The average precision of existing tools and CPS will be computed for five published simulations using truncation points of N set to 5%, 10% and 20% of the number of statements in the source code of the simulations. These truncation points are established and frequently used within the IR community (Salton 1983; Korfhage 1997). The published simulations that will be chosen will have the property that the average precision of each of the program statements can be determined analytically due to the structure of the simulation. The published simulations with this property will come from the fields of queueing theory, materials science, and physics (D'Orsogna and Chou 2005; Kleinrock and Gail 1996; Rios et al. 2009). Choosing published simulations with these properties allows the control or ground truth of the average precision of the program statements in the simulation source code to be identified efficiently and correctly.

4.2 MEASURES OF SUCCESS

1. ***CPS will be successful if 3/5 of the SMEs who participate in the MILCs report that CPS is more effective than any tool they have previously employed in facilitating their explanation and understanding of unexpected behavior(s) in exploratory simulations.*** The expectation is that all 5 of the SMEs in the MILCs will report that CPS is more effective than any tool they have previously employed. However, the 3/5 measure represents real progress, and it allows for the discovery of types of unexpected behaviors that will not benefit significantly from CPS.
2. ***CPS will be successful if 1) it has a higher average precision in an least one of the MILCs for all three truncation points than all of the existing tools and 2) it always at least matches the highest average precision of the existing tools in each of the five MILCs for each of the three truncation points.*** No existing tool repeatedly samples the variable states or program statements within dynamic program slices to determine the frequency with which they are executed; CPS will. Thus, for exploratory simulations that use random variables, CPS will be able to estimate the precision of each statement in the exploratory simulation for a given input. Due to this advantage, the expectation is that CPS will have a significantly higher average precision than existing tools for exploratory simulations that use random variables. This expectation is due to CPS' capability to rank and return the statements with the highest estimated precision. For those simulations that do not use random variables the expectation is that CPS will still perform as well as existing tools because it will capture the program statements that affect the unexpected behavior for each given input through dynamic program slicing. Recall, this standard of success does not measure the ability of the causal networks generated by CPS to facilitate SME understanding and explanation of an unexpected behavior. Instead, it measures the ability of CPS to improve the state of the art in retrieving program statements relevant to an unexpected behavior in exploratory simulations. Success under this measure will be responsible, in part, for CPS' success under the MILCs measure.

4.3 EXISTING TOOLS

CPS will be evaluated through the average precision measure and MILCs. In the MILCs evaluation each SME will identify the existing tools s/he previously used to facilitate an explanation of unexpected behaviors.

However, the SMEs will also be provided a set of existing tools to employ. Here each of these tools is described. Most of these tools are available as part of Eclipse (Eclipse Foundation 2009) or as a plug-in to Eclipse. Eclipse is a multi-language software development platform consisting of an integrated development environment (IDE) and a plug-in system to extend it. It is free to download and extend via the plug-in system (Eclipse Foundation 2009). Each of the existing tools CPS will be evaluated against for the average precision measure is also described here. These tools represent the spectrum of different methodologies for retrieving program statements relevant to a behavior: symbolic execution, static and dynamic program slicing.

1. ***Kaveri*** - Kaveri is a modular static program slicer for Java built using the Indus program analysis framework with an Eclipse-based user interface. Kaveri generates static program slices from within Eclipse along with an intuitive UI to view the slice (Jayaraman 2005). Kaveri will be used in the MILCs and average precision evaluation.
2. ***Eclipse Debugger*** - The Eclipse debugger is composed set of plug-ins included within Eclipse. It provides all standard debugging functionality, including the ability to perform step execution, to set breakpoints to inspect variables and set values, and to suspend and resume threads. Eclipse also has a special Debug view that allows users to manage the debugging or running of a program within the Eclipse IDE (Eclipse Foundation 2009). The Eclipse Debugger will be used in the MILCs evaluation.
3. ***DDState*** - DDState is an eclipse plugin that realizes Zeller's Delta Debugging approach (Zeller 2002; Clevle and Zeller 2005) to automated fault localization. DDState's inclusion in the evaluation set of existing tools represents the state of the art in automated fault localization. In (Zeller 2002; Clevle and Zeller 2005) the algorithm underlying DDState outperformed Jones et al.'s Tarantula (Jones et al. 2002), and Renieris's (Renieris and Reiss 2003) nearest neighbor approach to fault localization. DDState will be used in the MILCs evaluation.
4. ***Java Path Finder (JPF)*** – JPF is an open source Java virtual machine that symbolically executes a program for all possible paths for a given set of inputs (Havelund and Pressburger 2000). JPF tracks every program statement it symbolically executes to test the program for property violations like deadlocks or unhandled exceptions. It can be adapted to retrieve program statements relative to an unexpected behavior by returning the statements along all possible paths relevant to a particular program state for a given input abstraction. JPF will be used in the average precision evaluation.
5. ***JSlice*** - JSlice is a dynamic program slicing tool for Java programs. It collects and analyzes program statements in execution traces in a compressed form (Wang and Roychoudhury 2004). JSlice will be used in the average precision evaluation.

5. CONCLUSION

Every day public policy debates surrounding exploratory simulations such as Episims (Cha 2005) hinge on the ability to understand and explain unexpected behaviors. How can policy makers make informed decisions involving billions of dollars and millions of people in confidence when poorly understood behaviors in exploratory simulations are pervasive? Methodology to improve SME understanding and explanation of unexpected behaviors in exploratory simulations is needed; this motivates CPS.

CPS will automate the process of understanding unexpected behaviors with a methodology to construct causal networks of clusters of exploratory simulation executions. The causal networks will gather the insight that enables a SME explanation of an unexpected behavior. The evaluation plan will measure the effectiveness of this insight to SMEs and CPS' ability to retrieve program statements relevant to an unexpected behavior. All together the proposed research will provide SMEs with an improved methodology to facilitate the understanding and explanation of unexpected behaviors in exploratory simulations.

6. REFERENCES

- Agrawal H, DeMillo R A and Spafford E H (1993). Debugging with dynamic slicing and backtracking. *Software-Practice & Experience* 23(6): 589-616.
- Arthur W (1999). Complexity and the Economy. *Science* 284: 107-109.
- Atkinson D C and Griswold W G (2001). Implementation Techniques for Efficient Data-Flow Analysis of Large Programs. In *Proceedings of the IEEE International Conference on Software Maintenance* 16-27.
- Baciu A, Anason A, Stratton K, and Strom B (2005). *The Smallpox Vaccination Program: Public Health in an Age of Terrorism*. Washington, D.C.: Institute of Medicine of the National Academies.
- Bankes S (1993). Exploratory Modeling for Policy Analysis. *Operations Research* 43(3): 435-449.
- Bertoli M, Casale G and Serazzi G (2007). An Overview of the JMT Queueing Network Simulator. *Technical Report TR 2007.2, Politecnico di Milano – DEI*.
- Binkley D and Gallagher K B (1996). *Advances in Computers: Program Slicing*. Academic Press: St Louis.
- Brun Y, Ernst M D (2004). Finding latent code errors via machine learning over program executions. In *Proceedings of the 26th International Conference on Software Engineering* 480-490.
- Cha A (2005). Computers Simulate Terrorism's Extremes. In: Bennett P, Coleman M and Downie L (eds). *Washington Post July 4, 2005*. Washington Post: Washington, D.C., pp. A1.
- Chang B, Jo J, Her S H (2002). Visualization of Exception Propagation for Java Using Static Analysis. In *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation* 173-182.
- Clarke E M, Grumberg O and Peled D A (2000). *Model Checking*. MIT Press: Boston.
- Cleve H and Zeller A (2005). Locating Causes of Program Failures. In *Proceedings of the 27th International Conference on Software Engineering* 342 - 351.
- Darwhiche A (2009). *Modeling and Reasoning with Bayesian network*. Cambridge University Press: Cambridge.
- D'Orsogna M R and Chou T (2005). First Passage and Cooperativity of Queueing Kinetics. *Physics Review Letters* 95 170603.
- Eclipse Foundation (2009). *Eclipse*. Available at <http://www.eclipse.org/> accessed on May 20, 2009.
- Elder B, Dukic V and Dwyer G (2006). Uncertainty in predictions of disease spread and public health responses to bioterrorism and emerging diseases. In *Proceedings of the National Academy of Sciences* 103(42): 15693-15697.
- Eisenbarth T, Koschke R and Simon D (2001). Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of the 17th International Conference on Software Maintenance* 201-211.
- Eisenbarth T, Koschke R and Simon D (2003). Locating Features in Source Code. *IEEE Transactions on Software Engineering* 29(3): 210-224.

- Ernst M D, Perkins J, Guo P, McCamant S, Pacheco C, Tschantz T and Xiao C (2006). The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69(1-3): 35-45.
- Eubank S, Guclu H, Kumar A, Marathe M V, Srinivasan A, Toroczkai Z, and Wang N (2004). Modeling disease outbreaks in realistic urban social networks. *Nature* 2541(429): 180-184.
- Fortnow L (2009). The status of the P vs. NP problem. *Communications of ACM* 52(9): 78-86.
- Gallagher K B (1989). *Using Program Slicing in Software Maintenance*. PhD thesis, University of Maryland, College Park, MD.
- Gallagher K B and Lyle J R (1991). Using program slicing in software maintenance. *IEEE Transactions on Software Engineering* 17(8): 751-761.
- Ghezzi C, Jazayeri M and Mandrioli D (2003). *Fundamentals of Software Engineering*. Prentice Hall: Upper Saddle River.
- Gore R and Reynolds P F (2008). Applying Causal Inference to Understand Emergent Behavior. In *Proceedings of the 2008 Winter Simulation Conference* 712-721.
- Gore R and Reynolds P F (2009). Causal Program Slicing. In *Proceedings of the 23rd International Workshop on Principles of Advanced and Distributed Simulation* 19-26.
- Gore R and Reynolds P F (2009). INSIGHT: Understanding Unexpected Behaviors in Agent-Based Models. To appear in *Journal of Simulation*.
- Gore R and Reynolds P F (2009). Program Slice Distribution Functions. To appear In *Proceedings of the 2009 Winter Simulation Conference*.
- Havelund K and Pressburger T (2000). Model checking JAVA programs using JAVA Pathfinder. *International Journal on Software Tools and Technology* 2: 266-381.
- Hooke W and Pielke R (2000). *Prediction: Science, decision making, and the future of nature*. Island Press: Washington, D.C.
- Jayaraman G, Ranganath V and Hatcliff J (2005). Kaveri: Delivering the indus java program slicer to Eclipse. In *Proceedings of 2005 Fundamental Approaches to Software Engineering* 269-272.
- Jones J, Harrold M and Stasko J (2002). Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering* 467-477.
- Kalantarifard F, Nadagran H and Elahi P (2009). The analytical and numerical investigation of thermo-optic effects in double-ended-pumped solid state lasers. *International Journal of Physical Sciences* 4(6): 385-389.
- Kang B, and Bierman J M (1996). *Design-level cohesion measures: Derivation, comparison, and applications*, Technical Report CS-96-104, Colorado State University.
- Kleinrock L and Gail R (1996). *Queueing Systems: Problems and Solutions*. Wiley and Sons: New York City.

- Korel B and Yalamanchili S (1994). Forward computation of dynamic program slices. In *Proceedings of 1994 International Symposium on Software Testing and Analysis* 66-79.
- Korfhage R (1997). *Information Storage Retrieval*. John Wiley and Sons: New York City.
- Lakhotia A. and Deprez J C (1998). Restructuring programs by tucking statements into functions. *Information and Software Technology* 40(11-12): 677-691.
- Langville A N and Meyer C D (2006). *Google's PageRank and beyond: the science of search engine rankings*. Princeton University Press: Princeton.
- Law A and Kelton D (2000). *Simulation Modeling and Analysis 3rd Edition*. McGraw-Hill: New York City.
- Leon D, Masri W and Podgurski A (2007). An empirical evaluation of test case filtering techniques based on exercising complex information flows. In *Proceedings of the 27th International Conference on Software Engineering* 412 - 421.
- Liu C, Zhang X, Han J and Zhang Y, Bhargava B K (2007). Indexing noncrashing failures: A dynamic program slicing-based approach. In *Proceedings of the International Conference on Software Maintenance* 1-10.
- Mathworks (1999). *Using Simulink 4th Edition*. Mathworks: Boston.
- National Science Foundation (2006). *Simulation-based engineering science: Revolutionizing engineering science through simulation*. Report of the NSF Blue Ribbon Panel on Simulation-Based Engineering Science.
- National Science Foundation (2007). *Cyberinfrastructure vision for 21st century discovery*. Report of the NSF Cyberinfrastructure council.
- Padala P (2002). Playing with ptrace, part I. *Linux Journal* (103)5: 78-85.
- Padala P (2002). Playing with ptrace, part II. *Linux Journal* (104)4: 86-91.
- Pearl J (2000). *Causality: Models, Reasoning, and Inference*. Cambridge University Press: Cambridge.
- Planung Transport Verke (2005). *VISSIM User Manual 4.10*. Planung Transport Verkehr AG: Karlsruhe.
- Ranganath V P (2006). *Scalable and accurate approaches to program dependence analysis, slicing, and verification of concurrent object oriented programs*. Ph.D. thesis, Department of Computing and Information Science, Kansas State University.
- Renieris M and Reiss S (2003). Fault Localization with Nearest Neighbor Queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering* 30-39.
- Rios P R, Jardim D, Assis W L, Salazar T L and Villa E (2009). Inhomogeneous Poisson point process nucleation: comparison of analytical solution with cellular automata simulation. *Materials Research* 12(2): 219-224.
- Salton G and McGill M J (1983). *Introduction to modern information retrieval*. McGraw-Hill: New York.

- Shneiderman B and Plaisant C (2006). Strategies for evaluating visualization tools: multidimensional in-depth long-term case studies. In *Proceedings of the 2006 AVI workshop on BEyond time and errors: novel evaluation methods for information visualization* 1-7.
- Singhal A (2001). Modern information retrieval: a brief overview. *IEEE Data Engineering Bulletin, Special Issue on Text and Databases* 24(4): 1-9.
- Spirtes P, Glymour C and Scheines R (2000). *Causation, Prediction, and Search*. Springer-Verlag: New York City.
- Tip F (1995). A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3(3): 121-189.
- Trenouth J (1991). A survey of exploratory software. *The Computer Journal* 34(2): 153-163.
- Weiser M (1981). Program Slicing. In *Proceedings of the 5th international conference on Software engineering* 439-449.
- Whipple C (1996). Can nuclear waste be stored safely at yucca mountain? *Scientific America* 274(6): 72-79.
- Wilde N, Buckellew M, Page H and Rajilch V (2001). A Case Study of Feature Location in Unstructured Legacy Fortran Code. In *Proceedings of the 5th European Conference Software Maintenance and Reverse Engineering* 68-75.
- Wang T and Roychoudhury A (2004). Using compressed bytecode traces for slicing Java programs. In *Proceedings of the 26th International Conference on Software Engineering* 512-521.
- Wong W E, Gokhale S S and Horgan J R (2000). Quantifying the Closeness between Program Components and Features. *The Journal of Systems and Software* 54(2): 87-98.
- Xu R and Wunsch D (2005). Survey of clustering algorithms. *IEEE Transactions on Neural Networks* 16 (3): 645-678.
- Zeller A (2002). Isolating Cause-Effect Chains from Computer Programs. In *Proceedings of ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering* 1-10.