Causal Program Slicing

Ross Gore and Paul F. Reynolds, Jr.
University of Virginia
{rjg7v, Reynolds}@virginia.edu

Abstract

Unexpected model behaviors need explanation, so valid behaviors can be separated from errors. Understanding unexpected behavior requires accumulation of insight into the behavior and the conditions under which it arises. Explanation Exploration (EE) has been presented to gather insight into unexpected behaviors. EE provides subject matter experts (SMEs) with the capability to test hypotheses about an unexpected behavior by semi-automatically creating conditions of interest under which SMEs can observe the unexpected behavior. EE also reveals the interactions of identified variables that influence the unexpected behavior. Causal Program Slicing, improves EE by: automatically identifying all variables in the model that may influence the unexpected behavior, quantifying how the state changes in those variables influence the unexpected behavior, and mapping the quantified state changes in the variables to the statements in the model's source code that cause change in state. These capabilities require less SME knowledge and provide more insight than EE.

1. Introduction

Computational models are being used more and more to predict potential outcomes of systems involving human lives and costly resources. Generally when computational models are used to predict outcomes uncertainty exists about conditions affecting the system being modeled, and about the model itself. As a result, subject matter experts (SMEs) often experience unexpected program behaviors and must then explore whether the behaviors reflect an error or an unexpected behavior of the system being modeled.

Recently, the inability of researchers to explain the results of a developing computational model, Episims, has led to public policy debate. Episims models the nationwide spread of the smallpox virus under various vaccination strategies [5]. Previous established models 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 a targeted vaccination strategy is much more severe than 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" [2].

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 model developers cannot provide a clear explanation for the difference between their predictions under these vaccination strategies and previously established estimates [1]. Methodology to facilitate the understanding of the behavior of Episims and similar models is needed.

Our goal is to design and develop a novel approach to understanding model behaviors that allows SMEs to validate or reject unexpected behaviors efficiently, and with confidence. "Explanation Exploration" (EE) has been introduced [7] for demonstrating that a given unexpected behavior is valid. EE allows a SME to test hypotheses about the unexpected behavior as a modeled phenomenon is driven towards conditions of interest. Due to the complexity of models where unexpected behaviors frequently occur, the SME often does not know how to drive the model to conditions of interest directly. The term *conditions of interest* means when a specific condition of the modeled phenomenon is maximized, minimized or targeted to an exact point.

EE has been taken a step further by offering SMEs additional insight into the interactions of SME identified variables causing unexpected behavior in a model [8]. The causal inference portion of EE can also be applied to reveal the interactions of identified variables in the model which create the specified conditions of interest. This allows the SME to understand how the model was driven to create the specified conditions of interest.

Causal Program Slicing (CPS) is a program analysis technique combining program slicing and causal

inference that offers more insight into the interactions of model variables than EE with less SME supplied information. CPS offers the following capabilities to facilitate SME understanding of unexpected behavior: 1) the ability to automatically identify all the variables in the model that may influence the computation of the unexpected behavior, 2) the ability to capture state changes throughout model execution in each of the identified variables, 3) the ability to quantify how much influence each state change in a variable has on the unexpected behavior and 4) the ability to map each state change of each variable to the statement in the model's source code that caused the state change.

CPS is a significant improvement over the causal inference portion of EE. CPS does not require the SME to identify the variables in the model that influence the unexpected, EE does. CPS provides information about how each state of a variable influences the unexpected behavior. EE only provides information about how one state of a variable influences the unexpected behavior. Finally, CPS maps each quantified variable state change back to the statement in the model's source code that caused the variable to change state. EE does not have any notion of mapping variables to statements in the model's source code. These improved capabilities are the major contribution of our work.

Next, we review work related to and employed by CPS. Then we present CPS, describe how we applied it to a case study model and summarize our contributions.

2. Related Work

CPS draws on the areas of program slicing, sensitivity analysis, causal inference, design of experiments, program debugging. We review work in these areas and describe how the work relates to CPS.

2.1. Program Slicing

Program slicing is a decomposition technique that extracts program statements relevant to a particular computation within the program [18]. A program slice provides the answer to the question, "What program statements affect the computation of variable v at statement s?" [15].

Both static and dynamic program slicing exist. CPS only employs static program slicing. 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}.

```
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;
   else
12   x := sum;
13 print (x);
14 average := sum/n;
15 print (average);
```

(a)

```
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;
   else
12   x := sum;
13 print (x);
```

(b)

Figure 1: (a) The example program. (b) A static slice of the program using criterion {13 ,x}.

As shown in Figure 1(b), all computations not relevant to the final value of variable x have been "sliced away". Slices are computed by identifying consecutive sets of transitively relevant statements, according to data flow and control flow dependences [15]. Only statically available information is used to compute slices; hence, this is a static slice.

CPS employs static program slicing to automatically identify all statements in the model that may affect the unexpected behavior. This ensures that when CPS identifies the statements which have the strongest influence on the unexpected behavior, each statement that can affect the unexpected behavior will have been considered. Also, using static program slicing relieves the SME from having to identify variables which may affect the unexpected program behavior.

2.2. Sensitivity Analysis

Sensitivity analysis has been proposed as a methodology to explore the robustness of the behavior in a model [11]. The goal behind sensitivity analysis is to vary the initial parameters of the model by a small amount and rerun the model. This allows the SME to understand how sensitive the model is to parameters.

Sensitivity analysis gives the SME understanding about how variations in input parameters affect model outputs. Our goal is to move this analysis technique from the input-output level of the model to the source

code level of the model where model variables change state. This allows SMEs to understand how the variations in the input parameters change the model variables in the source code and how the state changes in the model variables influence model behavior. Performing this analysis at the source code level of the model allows us to quantify the influence of a variable's change in state on an unexpected model behavior or another variable's state. These quantified state changes can be mapped to the statements in the model's source code that cause the variables to change state. This approach to program analysis is the major contribution of our work.

2.3. Causal Inference

Causal inference procedures identify the causal structure of deterministic and stochastic systems. The procedures use the Causal Markov Condition to produce a causal theory explaining the cause-effect relationship of the variables of interest. The Causal Markov Condition is that "a variable X is independent of every other variable except X's effects conditional on all of X's direct causes [14]." A causal theory consists of a causal model and a set of parameters which specify how each variable is influenced in the causal model. A causal model is a directed acyclic graph, with a 1-1 mapping between vertices in the graph and variables of interest. A variable X is said to have a causal influence on a variable Y if and only if a directed path exists from vertex X to vertex Y in the causal model. The causal model serves as the basis for the causal theory. Each edge in the causal model is a 1-1 mapping with an element of the set of parameters associated with the causal theory. Each parameter specifies the strength of the causal influence (the absolute value of the correlation between X and Y) induced by the corresponding link [14], [12].

CPS employs causal inference as the sensitivity analysis mechanism to identify how strongly a given state of a variable influences the unexpected behavior or another variable's state. The strength of a causal influence is measured as the absolute value of the correlation coefficient between the two variable states. The absolute value of a correlation coefficient lies between [0, 1] inclusive. Using the correlation coefficient and conditional independence to measure causality comes from causal inference [14], [12].

Causal inference allows CPS to build chains of variable states which specify how the variable states influence each other and the unexpected model behavior. Each of these variable states can be mapped back to the statement in the model's source code which caused the variable to change state. This creates a chain of statements specifying how the statements influence each other and the unexpected behavior.

2.4. Design of Experiments

Design of Experiments (DoE) refers to experimental methods used to quantify indeterminate measurements of factors and interactions through observance of forced changes made methodically as directed by mathematically systematic tables [11].

CPS varies model input parameters to collect samples for each of the statements in the model's source code which can affect the unexpected behavior. This step of CPS relies on DoE models to efficiently and accurately configure the set of input parameters. The set of input parameters is used to execute the model to collect samples for each state of each model variable which may influence the behavior .

2.5. Delta Debugging

Delta Debugging [19] closely matches our goal of efficiently understanding the causes of anomalous program behavior. Delta Debugging is an automated approach to program debugging that isolates the causes of failing test cases systematically. A program run that passes a test case and one that fails the same test case are required to apply the algorithm. The cause of failure is isolated by assessing outcomes of altered executions of the program to determine changes in the program state that cause differences in test outcomes.

None of these techniques are applicable to stochastic software or relate how variations in input parameters cause program statements to determine the program output. Also, these techniques do not identify statements which have the strongest influence on a software behavior. CPS addresses each of these issues.

3. Causal Program Slicing (CPS)

Previously, "Explanation Exploration" (EE) [7] was introduced for exploring the possibility that a given unexpected behavior is valid. EE allows a SME to test hypotheses about an unexpected behavior as a modeled phenomenon is driven semi-automatically, employing COERCE optimization methods [17], towards conditions of interest. EE supports SME insight into interactions among identified variables causing

unexpected behavior [8], particurally the interactions which create the specified conditions of interest.

Causal Program Slicing (CPS) offers more insight into the interactions of model variables and source code statements than EE, and requires less SME input. CPS offers the following capabilities to facilitate SME understanding of unexpected behavior: 1) automatic identification of all variables in the model that may influence the computation of the unexpected behavior, 2) capture of state changes throughout model execution for each of the identified variables, 3) quantification of influence each state change in a variable has on the unexpected behavior and 4) mapping of each state change of each variable to the statement in the model's source code that caused the variable to change state. EE has been successful in assisting SMEs with understanding and validating unexpected behaviors [7], [8]. We believe the improved insight offered by CPS will provide superior SME assistance and yield even more successful results. We discuss the details of these improvements here.

3.1. Identifying and Preprocessing Statements

CPS begins with the SME identifying the state of the model that represents the unexpected behavior. The program statement in the model's source code at which this state can be observed is identified by its line number, L. The variable storing the value of interest related to the unexpected behavior is identified by the variable, V. Static program slicing is then applied using the static slicing criterion {L, V}. The static program slice will yield all the statements in the model's source code containing variables that may influence the state of the model representing the unexpected behavior. The static program slice ensures all statements containing variables that can influence the unexpected behavior are considered. Also it greatly reduces the number of statements that need to be considered in understanding the unexpected behavior. The list of statements in the static program slice is passed to the preprocessor.

CPS preprocesses model source code and inserts statements to capture state changes of variables in the static program slice. The inserted code collects the value of a variable in an identified program statement before and after the execution of the program statement. Thus state changes in program statement variables are mapped to model source code. The collected variables' values serve as samples which can be analyzed to determine the influence a variable's state has on the unexpected behavior or the state of another variable. Collection of values of variables as the variables change state throughout the model execution, and quantifying the influence the state changes in the variables have on the unexpected behavior or on each other is central to our work. Each quantified variable state changes can be mapped back to the statement causing the variable to change state.

Currently, three different types of program statements can be included in a static program slice, as depicted in Figure 2. This statement taxonomy is adapted from [13]. Any input statements included in the slice are treated as assignment statements to the variable storing the input value. Output statements are not included in the taxonomy because they cannot affect the value of an unexpected behavior. Conditional loop statements and conditional control-flow statements are very closely related and can be reduced to one another based on the definitions in Figure 2. However, separating conditional loop statements from conditional control-flow statements eases the explanation of how these statements are handled by the preprocessor. For the remainder of this section a variable on the left-hand side of an assignment statement is referred to as *y*, as in Figure 2.

Assignment Statement: $y \longleftarrow f(x_1,\ldots,x_n)$
where f is a function and $y, x_1,\ldots,x_n$ are variables

Conditional Loop Statement: **while** $p(x_1,\ldots,x_n)$ goto m
where p is a predicate, $x_1,\ldots,x_n$ are variables and m is a label

Conditional Control-flow Statement: **if** $p(x_1,\ldots,x_n)$ goto m **else** goto n
where p is a predicate, $x_1,\ldots,x_n$ are variables and m and n are labels

Figure 2: The three different types of program statements distinguished by the CPS preprocessor.

Assignment statements occurring outside of conditional loop statements or conditional control flow statements are the simplest case for the preprocessor to handle. The preprocessor inserts code that maps the value of *y* to the assignment statement, and the preprocessor inserts code that collects the value of *y* after the statement is executed.

Assignment statements that occur inside conditional loop statements but outside conditional control-flow statements are a more complex case for the preprocessor to handle. For each assignment statement with a *y* that is declared outside the scope of the conditional loop the preprocessor inserts source code that maps the value *y* to the assignment statement. The value of *y* is also mapped to the conditional loop statement the assignment statement is nested in. Then for each of the mapped statements the preprocessor

inserts source code immediately outside of the conditional loop to collect the value of each *y* after the entire loop is executed. The state of an assignment statement with a *y* declared inside the conditional loop does not exist outside the loop and thus does not need to be collected. The nesting of conditional loop statements is currently ignored in CPS. Assignment statements in nested conditional loops are handled as if they were assignment statements in non-nested conditional loops.

Assignment statements that occur inside a conditional control-flow statement but outside a conditional loop statement are a complex case as well. First, the preprocessor only considers those assignment statements with a *y* that is declared outside the scope of the conditional control-flow statement. Reasoning for this approach shadows that for the conditional loop statement case. Next, the preprocessor groups assignment statements with the same *y* located in different paths (if/else) of the conditional control-flow statement. The pairing ensures that the state of a variable is sampled regardless of the path taken through the statement during execution. A *y* in an assignment statement that cannot be properly grouped cannot be sampled. Otherwise an unequal number of samples for some variable states could occur.

If the conditional control-flow statement has more than two possible paths the preprocessor attempts to find a group of assignment statements. Each assignment statement in the group must come from a different path and size of the group must be equal to number of possible paths in the control-flow statement. This ensures a program statement is sampled for every possible execution.

For the assignment statements in conditional control-flow statements that can be properly grouped the preprocessor inserts code to map the variable *y* back to the group of assignment statements. The value of *y* is also mapped to the conditional control-flow statement the assignment statements are nested in. Next, the preprocessor inserts code to collect the value of *y* after each assignment statement is executed. Due to the grouping each assignment statement in the group will lie on a different path of the conditional branch. This approach handles nested conditional control-flow statements without issue. At each level of nesting the preprocessor applies the steps previously described. Code is inserted at each level of nesting to collect samples and perform the proper mapping for those assignment statements that can be properly grouped.

Assignment statements inside an arbitrary nesting of conditional loop and conditional control-flow statements are treated by backtracking and applying the previously described approach for each type of statement starting at the deepest level of nesting. However, the preprocessor only inserts code to map the program statements to *y* and collect the value of *y* when all levels of the nesting have been processed. This is due to the inability of CPS to process nested conditional loop statements. Remedies present opportunities for future work.

3.2. Model Execution and Causal Inference

Once the preprocessing step is complete the SME must identify a set of input parameters to explore. The set of input parameters will be varied to determine how changes in them change the state of the variables in the model's source code. We strongly encourage users of CPS to employ Latin hypercube sampling, orthogonal sampling, or another published sampling approach that provides efficient and equal density coverage of the search space for a given number of samples [10], [6].

CPS uses the set of input parameter configurations and the code inserted by the preprocessing to execute the model and collect the samples for each state of each variable in the static program slice.

Next, CPS quantifies the influence of a variable state on unexpected behavior or another variable state by applying causal inferencing to the samples of the variables' states. The result is a chain of variable states which specify how each variable state influences others, and the unexpected model behavior. Recall, the strength of a causal influence is measured as the absolute value of the correlation coefficient between two variable states. Using the correlation coefficient and conditional independence to measure causality is based on previous causal inference research [14], [11].

Using data stored by the preprocessor, each variable state that is over the user-specified threshold is 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 is displayed to the SME. The graph is annotated with the causal influence each program statement has on unexpected program behavior or another program statement over threshold. The graph focuses SME attention on understanding those statements in the model's source code with the strongest causal influence on the unexpected behavior.

CPS is configurable. The SME identifies the threshold causal influence a variable state must have on

the unexpected behavior, or on another above threshold variable state which influences the unexpected behavior to be included in the slice. Given a causal influence z, z has no influence if $0.0 \leq z < 0.1$, a weak influence if $0.1 \leq z < 0.3$, a moderate influence if $0.3 \leq z < 0.5$, and a strong influence if $0.5 \leq z \leq 1$ [3].

The SME also configures a maximum depth of function calls that CPS should search to identify those variable states which have a causal influence on the unexpected behavior.

3.3. Applying CPS to a Small Example

To help elucidate CPS, process we apply it to the program in Figure 1(a). The work of the preprocessor is shown in Figure 3. The `names` array maps statements to state changes in variables. The `samples` array records the value of the state change.

```
1  read(n);
   names[0] = "read(n);";
   samples[0] = n;
2  i := 1;
   names[1] = "i := 1;";
   samples[1] = i;
3  x := 0;
   names[2] = "x := 0;";
   samples[2] = x;
4  sum := 0;
   names[3] = "sum := 0;";
   samples[3] = sum;
6  while i<= n
7    sum := sum + i;
8    i := i + 1;
9  end
   names[4] = "while i<=n sum := sum + i; end";
   samples[4] = sum;
   names[5] = "while i<=n i := i + 1; end";
   samples[5] = i;
10 if (sum mod n == 0)
11   x := 1;
   else
12   x := sum;
   names[6] = "if (sum mod n ==0) x:= 1 else x := sum";
   samples[6] = x;
13 print (x);
```

Figure 3: The result of preprocessing Figure 1(a) with respect to the state of the variable x in line number 13.

CPS proceeds as follows:

1. The user identifies the value of `x` in line 13 as the program state capturing the unexpected behavior.
2. The user configures CPS to only collect those variable states within the same function that affect the value of `x` in line 13. The user also specifies a causal influence threshold of .6, this is the minimum influence a variable state must have on the value of `x` in line 13 or on another variable state which has an influence $\geq .6$ on the value of `x` in line 13.
3. The causal program slicer initiates static program slicing with the slicing criterion {13, x} to determine the program statements, shown in Figure 1(b) that influence the value of `x` in statement 13.
4. For program statements 1-4 the preprocessor inserts code to map the state of the variables `n, i, x` and `sum` back to their respective program statements. Code is inserted by the preprocessor to collect the state of each variable after each statement is executed.
5. The preprocessor identifies statement 6 as a conditional loop. The preprocessor inserts code to map this state of `sum` to statements 6, 7 and 9. Similarly, the preprocessor inserts code to map this state of `i` to statements 6, 8, and 9. Code to collect the state of `sum` and `i` is inserted after the end of the conditional loop in statement 9.
6. The preprocessor identifies statement 10 as a conditional control flow statement. It pairs the `x` in statement 11 with the `x` in statement 12 and inserts code to map this state of the variable x with statements 10, 11 and 12. Code to collect the state of variable `x` after the end of the conditional control-flow in statement 12 is added.
7. The user performs orthogonal sampling to generate 1,000 different values for input parameter `n,` and runs the program for each of the generated values. The values for `n` range between 1 and 10,000.
8. CPS performs causal inference on the generated samples. CPS outputs a causal graph including each variable state with an influence $\geq .6$ on the value of `x` in line 13 or on another variable state which has a causal influence $\geq .6$ on the value of `x` in line 13.
9. Each variable state is mapped back to the program statement that causing the variable to change state. The causal graph containing only the program statements with the strongest causal influence on the value of `x` in line 13 is shown in Figure 4.

The user gains insight from Figure 4. The initial value of `n`, and the state of `sum`, where the integers $\leq$ `n` are added together have the strongest influence on the value of `x` in line 13.

```
1  read(n);
       ↓ .9689
6  while i<= n
7    sum := sum + i;

9  end
       ↓ .6009
10 if (sum mod n == 0)
11   x := 1;
   else
12   x := sum;
       ↓ 1.000
13 print (x);
```

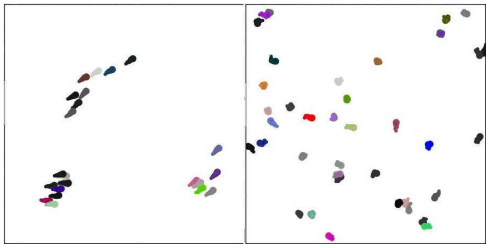
Figure 4: The causal program slice for Figure 1(a) with slicing criterion {13, x} and a threshold of causal influence of .6.

This example is not meant to be representative of an actual program with unexpected behavior, but to

illustrate how CPS works. Next, we present a case study where CPS is rigorously applied.

4. CPS and the Self-Driven Particle Model

To evaluate CPS we conducted a case study using a self-driven particle model [16], exhibiting unexpected behavior. In the self-driven particle model 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 when each follows the given set of rules. Figure 5(a) shows a typical model execution where three clusters have formed. Color is only used to distinguish particles from one another. However, under some parameterizations the particles exhibit a different behavior. Rather than joining a distinct cluster, each particle roams in a random walk. This behavior is called Spontaneous Symmetry Breaking [16], [9]. This is shown in Figure 5(b).



(a) (b)

Figure 5: (a) Clustering in the model. (b) Spontaneous Symmetry Breaking in the model.

It is claimed that the only model parameter with a strong influence on Spontaneous Symmetry Breaking is the amount of randomness applied to a particle's orientation [16], [9].

We apply CPS to the Self-Driven Particle Model to determine which input parameters influence Spontaneous Symmetry Breaking, how those input parameters cause state changes in variables in the model's source code, and how the state changes in those variables create Spontaneous Symmetry Breaking. CPS quantifies the influence of each variable state change on other variable state changes. Also, each variable state change can be mapped back to the statement in the model source code that caused the variable to change state. Thus, CPS reveals which statements in the model source code, beginning with input parameters, have the strongest influence on Spontaneous Symmetry Breaking.

The program state representing Spontaneous Symmetry Breaking is where the number of clusters in the model is computed. If particles are tightly clustered there will be very few clusters in the model. If particles are roaming the torus in a random walk there will be almost as many clusters are there are particles. Because many time steps of the model form a sample, we use the median number of clusters in the model across all the time steps. Line 354 captures the program state of interest. Figure 6 is the result of applying CPS with a causal influence threshold of 0.6.

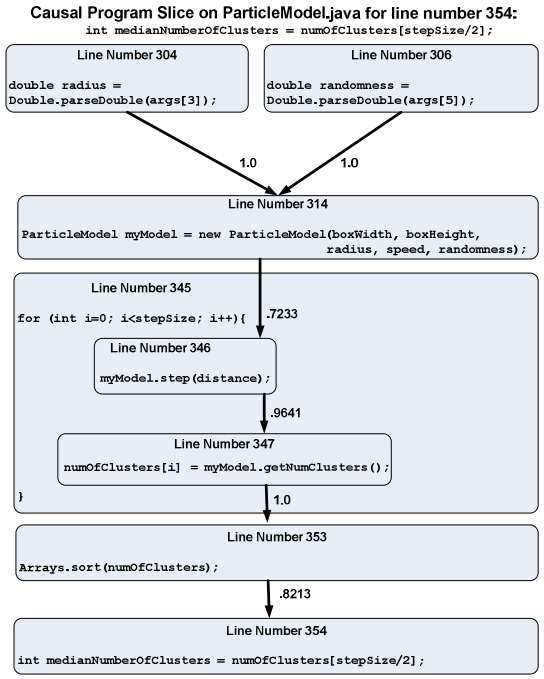


Figure 6: The causal program slice with threshold .6 with respect to line 354, the computation of the median number of clusters.

The function containing line number 354 comprises 57 lines of source code. Applying static program slicing to the program state representing the unexpected behavior, the computation of `medianNumberOfClusters` in line 354, reduces the number of statements in the source code that influence the unexpected behavior to 36. Applying CPS with a causal influence threshold of 0.6 to the computation of `medianNumberOfClusters` in line 354 reduces the number of statements to seven! Furthermore, the SME is guaranteed that these seven statements have the strongest influence on the

computation of the unexpected behavior.

Figure 6 shows that the two input parameters `randomness` and `radius` which are used to create the variable `myModel` have a strong causal influence on the number of clusters in the model. `randomness` is the amount of randomness in particle orientation and `radius` is the interaction radius around a particle. Recall, a particle identifies other particles within the interaction radius and chooses its own orientation based on the average of those particles.

Figure 6 also shows that the call to `step` (line 346) in the conditional loop statement changes the state of `myModel`. This call has a strong causal influence on the number of clusters in the model. The call to `getNumberOfClusters` uses the state change of `myModel` caused by `step` to change the state of `numberOfClusters[i]`.

Figure 6 reveals line 347 also has a strong causal influence on the number of clusters in the model. To better understand the influence of the variable states and program statements in the functions `step`, and `getNumberOfClusters`, on the number of clusters, we apply CPS again to each function. The result of applying CPS to the `step` function, with respect to the computation of `medianNumberOfClusters` in line 354, for a causal influence threshold of 0.6, is shown in Figure 7. Results of applying CPS to `getNumberOfClusters` are discussed later.

Figure 7 shows the state changes that occur in the `step` function that influence the number of clusters. Note particles are divided into clusters by the function `assignClusters` in `ClusterAssignment`. We apply CPS to the call to `assignClusters` which reveals that a state change to a variable in function `getNeighborsWithinRadius` has a strong influence on the number of clusters. This influence is a reflection of the causal influence of the radius parameter. Particles with a large interaction radius are able to identify more neighbors to include in a cluster, which limits the total number of clusters in the model and does not result in Spontaneous Symmetry Breaking. Conversely, a small interaction radius results in many clusters of a small number of particles and helps cause Spontaneous Symmetry Breaking. Figure 7 also shows that `assignClusters` returns an array which holds the number of particles in each cluster in the model. The array is sorted and `myModel.maxClusterID` is then set to the index of the last array component. This statement changes the state of `myModel.maxClusterID` to one less than the number of clusters in the model.

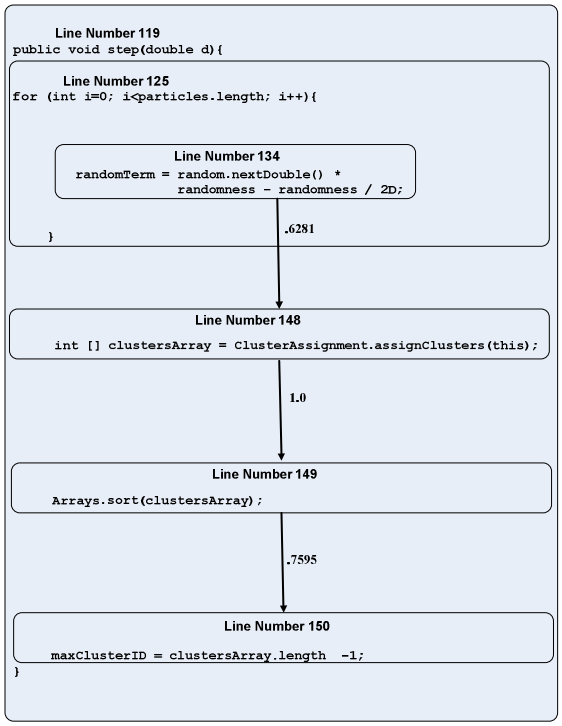


Figure 7: The causal program slice with a threshold of .6 for the function step with respect to line 354.

The state of `randomTerm`, a variable affected by `myModel.randomness` has a strong causal influence on the number of clusters. This is consistent with previous literature on self-driven particle models [18], [10]. A large random term does not allow the particles to organize into clusters, resulting in a large number of clusters. This reflects the causal influence of the model input parameter `randonmenss` on Spontaneous Symmetry Breaking.

There is only one state change that occurs in the `getNumClusters` function; `numberOfClusters` is assigned `maxClusterID + 1`. The effect of this state change on the number of clusters is already quantified, and explains the high causal influence `myModel.step` has on `numOfClusters[i]` in Figure 6. The function call `myModel.step` changes the state of `myModel.maxClusterID` and then immediately following that assignment the value of `myModel.maxClusterID + 1` is assigned to `numOfClusters[i]`. These two function calls count the number of clusters at each time step and store the result in `numOfClusters`.

The remainder of Figure 6 is fairly simple. The array

`numOfClusters` is sorted and then the median value is taken. Currently, CPS records the state of an array by taking its arithmetic mean. This is not generally ideal; improving this assumption is future work.

The claim of previous authors that randomness is the primary causal factor in determining number of clusters, while correct, has been demonstrated to be incomplete by our application of CPS. Other factors, in particular interaction radius, have significant influences themselves. A user equipped with the broader understanding of influences on clustering provided by CPS, possesses greater insight into the behavior of the model.

5. Conclusion

Our goal is to design and develop an efficient and effective approach to support SME understanding of unexpected model behaviors. We have improved the capabilities of the previously published approach, EE, by offering SMEs more insight into unexpected model behavior with less required SME information. CPS offers SMEs the following capabilities: 1) automatic identification of all variables in the model that may influence the computation of the unexpected behavior, 2) capture of state changes throughout model execution for each of the identified variables, 3) quantification of influence each state change in a variable has on the unexpected behavior and 4) mapping of each state change of each variable to the statement in the model's source code that caused the variable to change state.

6. References

[1] A. Baciu, A. Anason, K. Stratton, and B. Strom, *The Smallpox Vaccination Program: Public Health in an Age of Terrorism*, Inst. of Med. of Natl. Acad., Wash., D.C., 2005.

[2] A. E. Cha, "Computers Simulate Terrorism's Extremes", *Washington Post*, Washington, D.C., July 4, 2005, pp. A1.

[3] Cohen, J. *Statistical power analysis for the behavioral sciences*, L. E. Associates, Philadpelphia, PA, 1988.

[4] T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code", *IEEE Transactions on Software Engineering* 29:(3), March 2003, pp. 210-222.

[5] S. Eubank, H. Guclu, A. Kumar, M. V. Marathe, A. Srinivasan, Z. Toroczkai, and N. Wang, "Modeling disease outbreaks in realistic urban social networks", *Nature* 2541:(429), November 2004, pp. 180-184.

[6] A.G. Garcia, "Orthogonal sampling formulas: a unified approach", *SIAM Review* 42, September 2000, pp. 499–512.

[7] R. Gore, P. F. Reynolds, L. Tang, and D.C. Brogan, "Explanation Exploration: Exploring Emergent Behavior", *Proceedings of the 2007 Conference on Principles of Advanced and Distributed Simulation*, June 2007.

[8] R. Gore and P. F. Reynolds, "Applying Causal Inference to Understand Emergent Behavior", *Proceedings of the 2008 Winter Simulation Conference*, December 2008.

[9] C. Huepe, M. Aldana, "Intermittency and Clustering in a System of Self-Driven Particles", *Physical Review Letters* 92:(16), 2004.

[10] W. L. Loh. "On Latin Hypercube Sampling", *Annals of Statistics* 24:(5), 1996, pp. 2058–2080.

[11] Montgomery, D.C., *Design and Analysis of Experiments 6th Edition*, Wiley & Sons, Indianapolis, IN, 2004.

[12] Pearl, J., *Causality: Models, Reasoning, and Inference*, Cambridge Univ. Press, Camebridge, MA, 2000.

[13] S. Rapps, and E. J. Weyuker, "Selecting software test data using data flow information", *IEEE Transactions of Software Engineering* 11:(4), April 1985, pp. 376-375.

[14] Spirtes, P., Glymour, C., and Scheines, R. *Causation, Prediction, and Search*, Springer Verlag, NY, 2001.

[15] F. Tip, "A Survey of Program Slicing Techniques", *Journal of Programming Languages* 3:(3), pp. 121-189.

[16] T. Vicsek, A. Czirock, E. Ben-Jacob, I. Cohen, and O. Shochet, "Novel type of Phase Transition in a System of Self-Driven Particle", *Physicsal Review Letters* 75, 1995.

[17] S. Wazziruddin, P.F. Reynolds and D.C. Brogan. "The Process of Coercing Simulations", Proceedings of 2003 Fall Simulation Interoperability Workshop, Orlando, FL, 2003.

[18] M. Weiser, "Program Slicing", *Proceedings of the 5th Conference on Software engineering*, 1981, pp. 439-449.

[19] A. Zeller, "Isolating Cause-Effect Chains from Computer Programs", *Proceedings of Symposium on the 10th Symposium on Foundations of Software Engineering*, 2002.