Automatic Discovery of State

Assisting software maintainers through hueristic discovery of state features

Name Withheld per double blind review process
Affiliation Withheld
Location Withheld
Email@withheld.com

***Abstract*—Formal definitions of state have been used to great effect as a tool for testers, program verification, and in defect detection. Developers, however, create informal state-related elements far more frequently. Current state of the art does not afford opportunities to examine informal state-related program elements except at the cost of extensive human code review hours. This paper discusses a novel heuristic algorithm that automatically classifies all program elements with respect to state, patterns discovered by examination of the state space, and a set of secondary algorithms that detect and classify these patterns in a codebase—with a final saving of many human code review hours.**

Keywords-state; automation; heuristic; algorithm; classification

I. Introduction

Any program that does not maintain internal state is incapable of learning or remembering past experiences. Even a garage door opener must recognize its current state (e.g., OPEN) to correctly operate. A slightly more complex pocket calculator must remember multiple states, including the current screen value, the last button pushed, any previous answer, etc. While some elements of a program can and do serve their function without internal state, the program as a whole must maintain state between user actions (i.e., recalling all the characters I have already typed) as well as between executions (i.e., remembering my preference for the window size and location of the word processing program).

Although state is a basic concept taught to all novice programmers, more advanced concepts are taught upon that foundation: state machines, state patterns, etc. Well-designed software may contain state in key objects to avoid widely distributed and possibly conflicting state values. Dependency inversion, observer patterns, listeners, delegates, function pointers, and events allow one object to receive and act upon state changes monitored by a supervising object. These are some of the fundamental and most recognized software building blocks and yet they all depend on a rather informally defined concept: state.

In discussing state it is useful to distinguish between formal and informal state. Formal state, including finite state machines, state design patterns, and other reoccurring program elements are categorized as formal because they are critically engineered and difficult to casually change. It is, for example, a non-trivial exercise to add a new state or create a new edge between states in a finite state machine. Informal state, however, exists much more frequently and is added by developers to enable specific program features. An example of this might be that a specific button turns a verbose log on or off. The button somehow affects the state of the logger, and that state enforces one of two behaviors on the application. Formal state is well-documented and easily found, but informal state occurs much more frequently.

Formal state has been shown to be extremely useful to researchers for a variety of purposes. Automatic source code generation has been enabled by formal state [1], [2]. Software testing [3], [4], [5] and automated fault detection [1] have also been shown to benefit from formal state. Somewhat reversing the role, one researcher [6] derived a test coverage prediction capability from the same formal state machine that specified the application. Knowing the formal state defined in an application opens up windows of understanding into its workings.

Informal state should be just as interesting to researchers. Unfortunately, informal state is far less accessible. Developers may create a state machine, perhaps of very limited scope, without documenting its presence. As an undocumented feature, the state machine may grow by aggregation, be copied to create a series of related state features, or otherwise change over time. Without specific and current knowledge of the informal state, developers, maintainers, and researchers cannot subject it to the same kind of analysis supported by formal state.

There may be a wide gap between documented design and the *de facto* design embodied by a codebase [7]. From experiential evidence [7], the "systematic analysis necessary to create the formal documentation force[s the developer] to look at each routine carefully." This finding is consistent with Ratzinger [8], who found that frequent refactoring, a process that also requires systematic analysis, showed a negative correlation with code defects. Results like these support the generally accepted view [9], [10] that code reviews are the best, if not the only, way to discover severe defects in software.

Other research [11] has described a number of common error conditions directly related to state. These conditions include state variables that are never changed (i.e., the value remained constant), inter-related state, and redundant state. These conditions "were found to be the source of multiple complex bugs that were easily resolved by formalizing the state machine" but such formalization required "carefully tracking the state variables of each class", a process that would "increase the load on developers". Again, simply discovering the existence of undocumented or informal state in a codebase is a significant hurdle to overcome.

In various areas where human time is a limiting factor, heuristic algorithms have often proved useful. This strategy was employed successfully by [12], where model checking was guided by a heuristic algorithm to replace "considerable nonautomatic work by [human] experts". That case used a heuristic algorithm to identify counter examples and thereby prune unproductive portions of the search space. It is noteworthy that several different heuristic algorithms were employed and achieved different levels of success. This pattern, using various heuristic algorithms to approximate human evaluation of a problem space, is frequently used to alleviate the drain on human resources.

And thus we arrive at a problem. Informal state is likely to contain useful data that may be exploited to ascertain properties of a program and especially to prove its correctness. Furthermore, informal state is known to correlate with software bugs and anomalous behaviors that may be detected and resolved as a corollary benefit from the examination of state. Informal state, however, is not easily discovered without extensive human effort. The current state of the art lacks an automatic ability to replicate the human code review process and discover program elements associated with state.

This paper describes a novel method for automatically discovering program elements associated with state as well as the various state-related (anti)patterns that were observed. The paper is organized as follows: Section II describes the technique and tunable parameters for state discovery. Section III discusses the various state-related (anti)patterns that have been discovered using this technique. Section IV describes the methodology used to evaluate this novel method. Section V analyzes the data created during that evaluation. Finally, Section VI draws conclusions, discusses limitations of the approach, and suggests future work.

II. Automatic State Discovery

The initial goal of this research was to identify program elements associated with state and to accomplish this goal with near-human accuracy. In stating that goal, a key conflict may be recognized: the human definition of state is uncertain. Therefore, much effort was expended to more formally define state. That reasoning was eventually incorporated into an algorithm to classify all program elements according to their association with state.

Axiom I: State is evidenced by distinct and mutually exclusive behaviors. This statement should be self-evident even from the informal definition of state used in the previous section. If the state of the program has changed, then the behavior of the program must also change. Conversely, if the behavior of the program has changed, some element of state must be involved. Merely random fluctuations in behavior are not related to state and although multiple states may coexist independently, they must exist in separate state machines (i.e., the network state of my computer is independent of the save state of my document). In all cases where a program element may be classified as state-related, a change in behavior must be indicated.

Axiom II: Changes in program behavior are derived from control structures with diverse and mutually exclusive control flow paths. Control structures offer an obvious handle by which to correlate state-based behaviors with specific lines of code. In Figure 1 it is evident that mutually exclusive pathways exist for each control structure (e.g., ABD, ACD, EG, EFG, HJ, and HI*J). Each of these pathways defines a distinct behavior. As such, it is evident that control structures are the source of state behaviors.

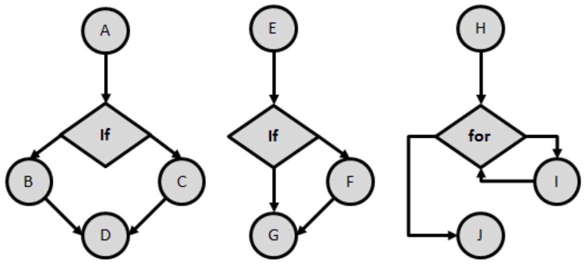

Figure 1. State behavior pathways in control structures

Axiom III: Any program element related to state will affect the evaluation of a control structure. This is the first axiom that was experimentally evaluated for correctness. Using an automated tool, 1408 variables and methods from multiple codebases were discovered to have been read or called in the evaluation of a control structure. Afterward, the complete set was manually evaluated for any relationship with state. The use cases inevitably included a variety of behaviors that were patently unrelated to state (e.g., null checks, comparison between values, checks for equality, and type checks), but these were initially included to offer the widest possible opportunity to detect state.

Of the 1408 variables classified, the majority (60%) were determined to be non-state and were most often used to prevent null pointers, validate data, order objects, check against min/max values, perform modulo arithmetic, and report successful completion of a subroutine. Almost without exception, the remaining state-related members were linked to Boolean or enumerated values. These results only used a binary flag to indicate if any number of control structure uses were associated with a program element, so the automated tool was altered to produce an incident count for each program element.

Comparing the new result with the manual classification revealed another useful trend. First, although many Boolean and enumerated variables were manually classified as state-related, these often had only one or two uses in a control structure. Second, the vast majority of the non-state program elements were used only once, while the highest incident numbers were exclusively allocated to members that had been manually identified as state-related. Finally, these incident counts were a somewhat reliable measure when considered for a full codebase. Individual classes sometimes had too few members to accurately distinguish state from non-state members. Therefore, in spite of an initial preference for the widest possible interpretation of state, the following conclusions were drawn from the experimental results.

Theorem I: Only direct evaluation of a program element in a control structure may be considered state-related. This theorem demonstrably creates a minor number of errors. According to this theorem, *if(b)* is a direct evaluation of b, while *if(b==true)* is an indirect evaluation. Because these two statements are logically equivalent, the exclusion of indirect evaluations may diminish the incident count for any variable incorrectly excluded by this theorem.

Theorem II: A high incidence of control-structure uses is indicative of state. This corollary to Axiom III can be used to exclude those program elements that only have a causal relationship to state. Under this theorem, Axiom III still holds true; however, the degree of association with control structures becomes a useful measure of "state-ness."

Theorem III: Any enumerated or Boolean variable is considered to be state-related as a native property of the variable type. The decision to include these types as "native state" is debatable; however, the experimental results showed that a large number of native state elements would otherwise be excluded by Theorem II.

Based on this logic, a combination of native-state and statistical methods were used to initially classify state-related members. A null hypothesis was generated from the contrapositive of Axiom III: Any program element with zero control uses is not a state variable. Thus, the incident count for each program element represents a deviation from the non-state hypothesis. These incident counts were used to calculate the standard deviation (see Equation 1) for all program elements.

$$(1)\ \sigma = \sqrt{\frac{1}{N}\Sigma(x_i - 0)}$$

Initially, it was hoped that a relatively high confidence interval might be used to draw a meaningful distinction between state and non-state members. Using a Z value of 1.75 as a cutoff value, however, still left too many non-state variables. The incident count from the 1408 members was plotted at 2, 3, 4, and 5 times that interval to create the graph in Figure 2. The graph shows the portion of non-state variables and methods decreases sharply between 2 and 3 times the intended interval. The data set indicated that a high-incident cutoff occurred around 2zσ or 3.5σ. Considered individually for each codebase in the dataset, the same relationship held true.

One final observation from the manual classification experiment seemed relevant. A number of native state members had zero control structure uses, but they had accessors that had been classified as state-related using the nzσ algorithm. Also, some abstract or interface methods had high incident counts while their implementations in child classes had zero incidents. These anomalies suggested two

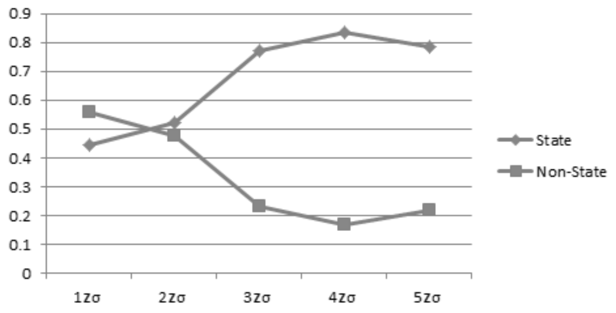

Figure 2. State and Non-State portions at nzσ

ways in which state-ness was a shared property between related program elements. By tracing relatively simple links between these elements, supposedly non-state members might be included because of their ready association with state members.

These considerations formed a fairly comprehensive map of state-related program elements based on their observable characteristics. In summary, a field or method is considered to be state-related if it meets one or more of these conditions:

1. The type of the field is Boolean or Enum. These are referred to as "native" state fields.
2. The field or method is used in control structures more than 3.5σ above the known non-state variables. This is known as the nzσ algorithm and these members are "statistical" state fields or methods.
3. The field or method may be directly linked to a field or method that has a direct data-flow or inheritance path to an existing state-related member.

A. Sources of error

Naturally, this classification system provides a number of opportunities for type I and type II errors. As noted earlier, Theorem I may incorrectly lower incident counts increasing false negatives, while Theorem III almost certainly introduces false positives. Furthermore, the data in Figure 2 indicates a 15–20% false positive rate even among those members classified by the nzσ algorithm. These sources of error are to be expected of heuristic algorithms, which by definition are not demonstrably correct. As to the reliability of the system, the reader is invited to consider the results of experimental evaluation described in sections IV and V.

B. Parameterization

As stated previously, the initial goal was to produce a classification with near-human accuracy. This classification system has an additional benefit in that the nzσ algorithm may be parameterized to change sensitivity. The values described here were selected to reduce possible false positives while retaining a reasonable degree of sensitivity, but the value of n may be altered to produce different effects. Higher values of n will have fewer false-positives and more false-negatives. This offers the user an opportunity to consider various levels of sensitivity, an ability that may prove valuable in some circumstances.

C. *Framework Limitations*

Much more will be said about the limitations of this technique in general in Section VI, but several limitations were derived from the automated tool itself below the level of the classification system. That automated tool was based on a Java 7 ANTRL grammar that limited the language and age of programs that could be examined. In addition, the framework was unable to identify program elements accessed using the Reflection API, using generics, or within collections. Finally, the tool did not use the full Java library during parsing and compilation, preferring instead to make assumptions about classes that were "external" to the codebase under examination. These limitations sometimes produced unusual results demonstrated by the code sample in Figure 3. Here, a list of objects is received and state is set for each object in the list. Because of these limitations in the underlying framework, however, *setState* is assumed to be a method of the external class *List*. While such errors on the part of the framework limited the effectiveness of the classification system, the errors represent a lower bound on error rather than an artificial inflation. Indeed, improvements to the framework should improve the results discussed below.

```
List<SomeObject> solist = getMyList();
for(int i=0;i<solist.count;i++)
{
  solist[i].setState(value);
  //true call to SomeObject.setState
  //tool interprets as List.setState
}
```

Figure 3. Sample code incorrectly parsed by tool.

III. State-Related Anitpatterns

The manual classification discussed in Section II also served to identify patterns with positive and negative consequences. These patterns are comparable to those found in [11], and secondary classification algorithms were designed to further identify areas of apparent low or high quality. In all, seven classification algorithms were built on top of the basic classification system. The patterns and antipatterns identified by these algorithms are briefly discussed here.

Setting. This pattern is considered neutral. A setting is a state variable that is set only once for the lifetime of an object instance. Such a variable might be set during construction or from a default value overridden by some of the constructors. Frequently, this pattern created two or more distinct sets of object instances, each of which expresses a fixed state.

Setting logic. This pattern is considered neutral. Any method that contains a control structure dependent on a setting should be considered further by a developer. Certainly, some logic allows instances with distinct settings to be uniformly handled by a single method while expressing different behaviors. Occasionally, however, a developer may incorrectly assume a variability that does not exist. If the logic is monitoring for a change in the setting, that logic is almost certainly flawed.

Constant and Constant logic. These patterns are considered negative and were discussed in [11], where code examples showed that constructors, accessor methods, and other logic may be used to set the variable, but always to the same value. The result is an apparent variable, but one that in fact does not change state. As with settings, logic that depends on a constant is suspect, often representing a tautology or contradiction and uniformly enforcing a single control path through the code. This condition is hard for a maintainer to detect, but is universally incorrect.

Setting sources. This pattern is considered positive. State variables are often derived from user-defined preferences stored as user setting files, read from installation properties, and in similar repositories. The objects that read in and disseminate this information are setting sources, and a detailed examination of those objects may be instructive to a novice developer (i.e., one who is unfamiliar with the codebase). Often these classes give clues as to the operation of the application, and following the dissemination of settings throughout the codebase will identify all areas of the code related to specific settings.

Super-state methods. This pattern is considered positive. Some state-related methods are introduced as part of an interface or abstract class. Whether a default implementation is given or not, each child class that overrides the method is inheriting a relationship to state. The expression of state may be unique for each class in a hierarchy or may be expressed for sub-groups. Linking one state-related method to all implementations in a hierarchy may also prove instructive to the novice programmer. Often these methods reveal a fundamental state relationship that is widely replicated throughout the codebase. Such a feature is fundamental to the design and is worth discovering in this way.

Stateful objects. This pattern is considered positive. Not to be confused with the State Pattern [13], these objects contain not just one state variable but several independent ones. They do not necessarily enact any behavior themselves, but these objects often encapsulate a state reference for one or more unrelated classes. Stateful objects represent an encapsulated nexus of state and, when coupled with the observer patter, may be used to transmit state from one area of the program to another driven by user actions and events.

Unread native state. This pattern is considered negative. As noted above, Theorem III sometimes classifies a native state variable that may not be used in a controls structure. After linking, some relation to state should become evident. Occasionally, however, a field still remains unread. These cases are compounded by a network of setter, constructor, and other method calls that feed state values into the variable. This network of supporting methods may be quite extensive and indicate either a long-standing relationship that is no longer valid or an erroneous assumption by maintainers. After examining the unread native state variable, removal of the variable along with many sections of related code may be required.

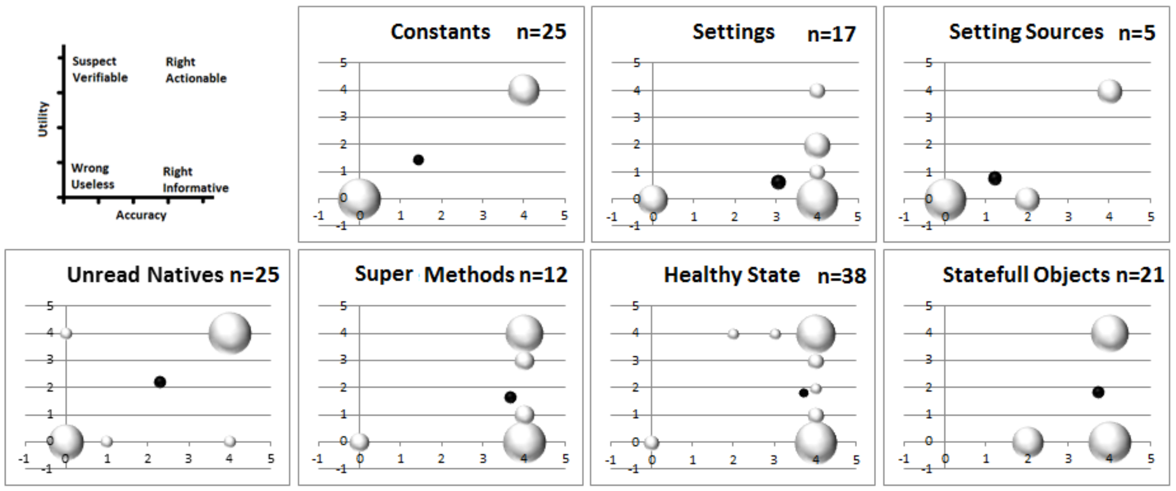

Figure 4. Experimental results

IV. Evaluation

In keeping with the stated objective of this research, the classification system and secondary classification algorithms were evaluated in comparison with human classification. Six open-source and commercial software projects were used for this purpose. For each codebase the various algorithms were used to discover and classify state-related members and the seven state patterns. For each of the seven categories, a statistical sample of the results was randomly selected for manual classification by an experienced developer who was nonetheless inexperienced with the specific software project.

Each result was graded for accuracy and utility. Accuracy was assigned an integer value according to the developer's confidence in the report ranging from 0 (no confidence in the report) to 4 (absolute confidence in the report). As a result of confirming or disproving the report, the developer determined if some specific course of action was called for. Often the course of action was obviously related to the report (e.g., remove a constant and the related logic), but the developer was free to note other actions directly identified through an examination of state (e.g., simplification of a state machine throughout a set of related classes). Again the developer assigned an integer value for utility ranging from 0 (no obvious action required) to 4 (immediate action required). The full data set is found in Table 1 at the end of this document.

As shown in Figure 4, the resulting data was plotted as a bubble graph on the plane of two axes showing accuracy vs utility. The size of the white bubble graph markers indicate the relative portion of the results that were found at each value. Individual markers are referenced by their integer components in an ordered tuple, such as <3,1> indicating a 3 for accuracy and a 1 for utility. In the top right corner of each graph is the size of the evaluated result set (n). The sample size varied greatly depending on the number of results reported for each codebase. Some report categories, like setting sources, were relatively rare when compared with the baseline healthy state variable discovered by the classification system.

In addition to the white markers indicating the cumulative data across the six sample programs, the graphs include a black marker. This is the centroid of the dataset and may be used to represent the total influence of all data points. Any apparent size discrepancy in the black marker is due to the limitations of the graphics and is not statistically significant.

V. Analysis

As with all automated metrics, false positives should be a first concern. False positives lead maintainers to doubt the accuracy of the tool and reinforce the behavior or ignoring all reports [14]. This classification system, however, was parameterized specifically to reduce the number of false positives and directly avoid eroding belief in the results. Unfortunately, such a decision has an adverse effect on false negatives that, while potentially important, are not easily tracked and were not in fact counted as part of the experimental protocol. From the earlier discussion, it would seem likely that both false positives and negatives will occur as a result of these algorithms

In the graphs in Figure 4, false positives <0,x> do occur. Based on multiple sources [15], [14], a reasonable standard for an acceptable false positive rate is 30% with truly "stable" algorithms producing false positives less than 20% of the time. Taken individually, three of the algorithms have false positive rates in excess of the acceptable limit: constants (64%), source objects (60%), and unread natives (40%). The algorithm to classify settings was the only one to fall in the 20–30% range. The remaining algorithms are well below the 20% range, indicating they are highly stable. The cumulative error among all reports was only 25%, well within the acceptable level.

Based on evaluation comments, it appears developers can learn to predict and even accept the limitations of the metrics because of that very predictability. As an example, fully half of all false positives were caused by the previously discussed limitations of the automated tool framework, not the proposed algorithms. The comments from developers recognized the source of those limitations and, while rapidly dismissing individual reports, they appeared to accept the predictable blind spots the tool had.

Furthermore, although some of the results from secondary classification algorithms (i.e., constants) were identified as false positives, the comments reveal that the primary identification with state was still accurate. Take as an example the developer remark for the unusual <0,4> report for an unread native. "Use wasn't detected because of a collection, BUT search revealed that two classes share the state variable and should probably be combined in some hierarchy." The comment shows that the native variable was read, contrary to and adversely affecting the need for accuracy, but the variable was still state-related and that a brief examination of the state space revealed a need for improvement of the code in this area. For similar reasons, a number of false positives only indicated that the state variable involved had been incorrectly classified as a constant, setting, etc.

While not intending to excuse the false positives in any sense, it appears the level of false reporting is acceptable for a variety of reasons. It appears that the basic classification of healthy, state-related members is highly reliable (95% accurate) but that secondary classifications were less accurate. Improvements to the metric framework may significantly decrease the false positive rate, no doubt a subject for future work.

It is also worth noting that the utility of these findings varies greatly depending on the positive, neutral, or negative connotations of the pattern. Settings, a neutral pattern, rarely produced actionable information, but may be seen as informative. On the other hand true positive reports for constants, a decidedly unwanted feature, were universally actionable. Most interesting of all is the dichotomy in reports for healthy state and stateful objects.

Healthy state and stateful object reports are considered positive or at least neutral, but some actionable information was found in roughly half of the reports. This suggests the automatic discovery of state members produced a significant return on investment. Having successfully replaced human hours with automatic classification of state-related members, even the fairly benign reports appear to have significant value in reshaping, improving, and fixing the codebase.

Finally, the developer comments reveal another point of interest that was discovered during evaluation. Among the six codebases were two reports that proved to be the root of known bugs. In one case, a verbal description and demonstration of the tool demonstrated that a window could not be resized. In another case, comments in the code itself showed that another GUI element could not retain its position in the foreground. Both were known bugs, but the root cause of the problems had never been determined. In just a few minutes of developer time, the examination of these two reports identified flaws in the state machine that were directly related to these bugs.

This last finding demonstrated that automated systems were able to replicate the discovery of state-related bugs that had previously been detected by [11] using manual processes. As the express purpose of this research was to replicate identification of state members with near-human accuracy, this finding should be considered conclusive evidence of success. Certainly much more can be done to improve these algorithms; however, the basic task of automatically detecting state-related members has been successfully completed.

VI. Conclusions and Future Work

This paper has described a system for automatically classifying program elements related to state using a combination of logical reasoning and experimental results. The paper further demonstrates by experimentation that the state classification system is 95% accurate when compared with human classification of the same program members. This classification system is parameterized with respect to sensitivity and may be altered to meet specific user requirements with respect to the balance of type I and type II errors.

Furthermore, the paper discusses a set of secondary classification algorithms meant to identify specific patterns in the state space. These algorithms were not found to be as accurate as the basic state classification, but may be elevated in accuracy through specific improvements to known limitations and weaknesses in the automated tool that supports the various classification algorithms.

These findings suggest that by using these techniques, a tool may support software maintainers in the automated discovery and evaluation of informal state structures that exist as undocumented features in a given codebase.

Because of the limitations in the underlying tool framework, these experiments were conducted for programs written in the Java programming language. Nevertheless, the algorithms are independent of Java language features and are consistent with general object-oriented program features. While replicating these experiments in another language would require significant rework in the tool framework, the experiment should be readily repeatable for any program written in an OO language.

Future work in this area naturally includes replicating the experiment for other OO languages as well as improving the algorithms to avoid specific conditions detected as false positives during this experiment. In addition, the parameterized element of the nzσ algorithm bears further scrutiny. For example, the algorithm may be repeated for a large set of n values to determine the sensitivity of all program elements to n. Additional opportunities to explore state space may be indicated by the calculated n-sensitivity values. Finally, third-party tools may be built to rely on this state discovery technique. What those tools may detect, report, or investigate, as well as the effect of informal state discovery on the areas of software testing and software verification remains to be seen.

Acknowledgments

Full acknowledgement withheld for the purposes of blind review process.

References

[1] J. E. Walker, “Generic software machine”. U.S. Patent 6,138,171, 24 October 2000.

[2] R. A. Kita, M. E. Trumpler and L. S. Elkind, “Method and apparatus for generating an extended finite state machine architecture for a software specification”. U.S. Patent 5,870,590, 9 February 1999.

[3] T. S. Chow, “Testing software design modeled by finite-state machines,” *IEEE Transactions on Software Engineering,* Vol. SE-4, no. 3, p. 178–187, May 1978.

[4] B. A. Curtis and M.-D. J. Hsu, “System, method, and program for automatic error detection while utilizing a software state machine for carrying out the process flow of a software program”. U.S. Patent 6,397,355, 28 May 2002.

[5] G. Friedman, A. Hartman, K. Nagin and T. Shiran, “Projected state machine coverage for software testing,” in *ISSTA ‘02 Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, Rome, Italy, 2002.

[6] J. Blom, A. Hessel, B. Jonsson and P. Pettersson, *Specifying and Generating Test Cases Using Observer Automata (Formal Approaches to Software Testing. FATES 2004),* vol. 3395, J. Grabowski and B. Nielsen, Eds., Berlin: Springer, 2005.

[7] D. L. Parnas, “Software aging,” in *Proceedings of 16th International Conference on Software Engineering*, Sorrento, Italy, 1994.

[8] J. Ratzinger, T. Sigmund and H. C. Gall, “On the relation of refactorings and software defect prediction,” in *Proceedings of the 2008 international working conference on mining software repositories*, Liepzig, Germany, 2008.

[9] F. Bourquin and R. K. Keller, “High-impact Refactoring Based on Architecture Violations,” in *11th European Conference on Software Maintenance and Reengineering (CSMR’07)*, Amsterdam, the Netherlands, 2007.

[10] S. Wagner, J. Jürjens, C. Koller and P. Trischberger, “Comparing bug finding tools with reviews and tests,” in *Proceedings of the 17th IFIP TC6/WG 6.1 International conference on Testing of Communicating Systems*, Montreal, Canada, 2005.

[11] Anonymous Authors. Omitted per double blind reviewing.

[12] A. Groce and W. Visser, “Heuristics for model checking Java programs,” *International Journal on Software Tools for Technology Transfer (STTT),* vol. 6, no. 4, p. 260–276, 2004.

[13] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design patterns: elements of reusable object-oriented software, Boston, MA, MA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[14] A. Bessey, K. Block, B. Chelf, A. Chour, B. Fulton, S. Hallem, A. Kamsky, S. McPeak and D. Engler, “A few billion lines of code later: using static analysis to find bugs in the real world,” *Communications of the ACM,* vol. 53, no. 2, p. 66–75, February 2010.

[15] T. Kremenek, A. Y. NG and D. Engerl, “A factor graph model for software bug finding,” in *Proceedings of the 20th international joint conference on artifical intelligence*, Hyderabad, India, 2007.

Table 1. Evaluation data set in ordered pairs (accuracy, utility)

	Constants		Setting		Setting Source		Unread Native		Super Method		Healthy State		State Object	
Application 1	0	0	4	0	0	0	4	0	4	1	4	4	4	0
	0	0	4	0	4	4	4	4	4	3	4	0	4	0
	0	0	0	0	0	0	0	0	4	4	4	0	2	0
	0	0	4	0	0	0	4	4	4	4	0	0	4	0
	0	0	0	0			0	0	4	4	4	3	2	0
	4	4	0	0			4	4	4	4	4	3		
	4	4									0	0		
Application 2	0	0	4	0	2	0	4	4	4	0	4	0	4	4
	0	0	4	2			4	4			4	4	4	4
	4	4	4	2			4	4			4	0	4	4
	4	4	4	1			4	4			4	0		
			4	2			1	0			4	0		
							4	4			2	4		
											3	4		
Application 3	4	4	4	0			0	0			4	4	4	4
	4	4					0	0			4	1	4	4
							4	4			4	1		
							4	4			4	4		
Application 4	0	0	4	0			0	0	4	0	4	0	4	0
	4	4	4	0			0	0	4	0	4	0	4	0
	4	4	0	0			0	0	4	0	4	4	4	0
	0	0	4	0			0	0	4	0	4	0	4	0
	0	0	4	4					0	0	4	0	4	0
											4	0		
											4	4		
Application 5	0	0									4	4	2	0
	0	0									4	4	2	0
	0	0									4	4	2	0
	0	0									4	4	4	4
	0	0									4	4	4	4
	0	0									4	4		
											4	4		
Application 6	4	4					4	4			4	0	4	0
							0	0			4	0		
							4	4			4	0		
							4	4			4	0		
							0	4			4	2		
											4	0		
Centroid	1.44	1.44	3.0588	0.6471	1.2	0.8	2.28	2.24	3.6667	1.6667	3.7105	1.8421	3.5238	1.3333