Helping Users Avoid Bugs in GUI Applications

Amir Michail
University of New South Wales
Sydney, NSW, Australia, 2052
amichail@cse.unsw.edu.au

Tao Xie
University of Washington
Seattle, WA, USA, 98195
taoxie@cs.washington.edu

Abstract

In this paper, we propose a method to help users avoid bugs in GUI applications. In particular, users would use the application normally and report bugs that they encounter to prevent anyone — including themselves — from encountering those bugs again. When a user attempts an action that has led to problems in the past, he/she will receive a warning and will be given the opportunity to abort the action — thus avoiding the bug altogether and keeping the application stable. Of course, bugs should be fixed eventually by the application developers, but our approach allows application users to collaboratively help each other avoid bugs – thus making the application more usable in the meantime. We demonstrate this approach using our "Stabilizer" prototype. We also include a preliminary evaluation of the Stabilizer's bug prediction.

1 Introduction

Despite advances in testing and formal verification, applications today are still plagued with thousands of bugs. For example, as of this writing, the Mozilla browser has around 20,000 open bugs. Bugs can be annoying in various ways. A crash is disastrous if the user has not saved his/her work recently. A bug may be more subtle, slowly corrupting state — and perhaps corrupting saved files also. Moreover, bugs could simply be wrong, unexpected, or intuitive behavior, often frustrating the user.

The situation is even worse than this: not only do applications today contain thousands of bugs, but many of those bugs linger for a long time. For example, bugs in the Linux kernel have an average lifespan of 1.8 years, with the median being around 1.25 years [2].

Given that bugs are a fact of life today and that they often remain unresolved for a long time, what can we do to improve this situation? Could users somehow avoid bugs altogether — and the annoyances that come with them — by simply avoiding the situations that trigger them? Avoiding bugs would make the application more usable until those bugs are eventually fixed.

The idea of avoiding bugs is not new: it is already done manually by users. Anyone who has encountered a bug will likely try to avoid it in the future. But such a manual approach has problems.

First, the manual approach to bug avoidance requires remembering the bug, which can be difficult if the application has many bugs. Moreover, this memory burden is actually worse than it sounds, since the user needs to remember new bugs on each release. If a bug is fixed, the user would also want to remember that as well to take advantage of a previously broken feature. And of course, if the user is using many applications, then the user needs to remember bugs (and whether they are fixed yet) for each of those applications.

Second, the manual approach to bug avoidance does not make it easy for users to learn from other users. For example, it would be better if a user could avoid a bug *without even encountering it once*. This could be done if a user finds out about the bug in advance from other users. But this rarely happens. It is completely unrealistic to expect a user to read through and remember thousands of bugs in a bug tracking system so that he/she could avoid them — particularly if the user is likely to encounter only a small fraction of those bugs in his/her typical usage.

Third, the manual approach to bug avoidance requires the user to figure out the circumstances under which a bug occurs. But manually identifying the bug exposure conditions may not be easy. Moreover, for particularly complicated bug exposure conditions, there is much to gain by pooling together execution context from many users to determine the contexts in which a bug occurs.

In this paper, we propose a tool-based approach to help users avoid bugs in GUI applications. The idea is that users would use the application normally and report bugs that they encounter to prevent anyone — including themselves — from encountering those bugs again. When a user attempts an action that has led to problems in the past, he/she will receive a warning and will be given the opportunity to abort the action — thus avoiding the bug altogether and keeping the application stable.

Observe that such a tool-based approach directly addresses the three problems of manual bug avoidance mentioned earlier. First, since the user would be given a warning that a bug is likely, the user need no longer remember bugs to avoid them. Second, since these bug warnings would be based on bug reports from all users, it is now possible for a user to avoid a bug without even encountering it once. Third, bug exposure conditions can be determined from bug reporting information from multiple users using automated methods (e.g., machine learning techniques).

We have built a system to demonstrate our approach, which we have named "Stabilizer", as avoiding bugs can be seen as "stabilizing" the application. Currently, the Stabilizer works with Java GUI applications.

The rest of the paper is organized as follows. Section 2 motivates our approach by example. Section 3 describes the bug prediction method. Section 4 presents a preliminary evaluation of bug prediction. Section 5 discusses related work. Section 6 summarizes the paper and suggests future work.

2 A Motivating Example

To motivate our approach, we shall run a *target application*, FreeMind 0.7.1, under the Stabilizer to demonstrate what typically happens when a bug is encountered. FreeMind is a "mind-mapping" tool written in Java, consisting of 21,983 lines of code. It allows users to easily create and modify a visually pleasing tree of concepts (e.g., for brainstorming).

In what follows, we shall describe step-by-step what would happen if we perform a certain sequence of actions in FreeMind starting with an empty Stabilizer database.

First, we start up the Stabilizer system. Currently, this is done as follows: (1) start up the *Stabilizer server* then (2) start up the *Stabilizer client*. The Stabilizer server is responsible for maintaining a central database of bug avoidance data. The Stabilizer client provides the GUI to the Stabilizer system and is responsible for making use of bug avoidance data for bug prediction. (In Figure 1, the client is the window on the left.) Second, we start up the target application, FreeMind, using the *Stabilizer runner*. (In Figure 1, the target application, FreeMind, is the window on the right.) The Stabilizer runner does on-the-fly Java bytecode instrumentation of the target application and will abort GUI callbacks if asked to do so by the client.

When an application is started with the Stabilizer runner, the Stabilizer client will download bug avoidance data from the Stabilizer server. (In this example, the server database is empty at application startup.) When the user reports a bug (or "not bug" as explained below), the client keeps track of this new bug avoidance data. The client is also responsible for making bug predictions based on the data downloaded from the server and also any additional bug avoidance data accumulated in this particular application execution. When the user quits the application, the client updates the server database with the new bug and "not bug" reports.

Returning to our example, when we start up FreeMind under the Stabilizer, we perform the following actions: (1) we create a new mind map (which automatically gets a root node with text "New Mindmap"); (2) we change zoom level from 100% to 200%; (3) we give the root node a child by pressing F10 to access the menu, then using the keyboard to select menu item Edit⇒New Child Node; and (4) we type "a" as the text for the newly created child node. So far we have not observed any bugs. (Figure 1 shows the results of the actions performed thus far.)

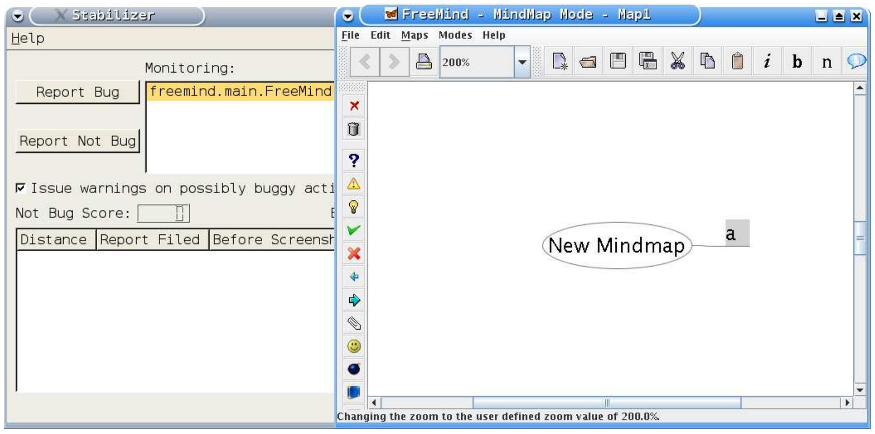
Next, we attempt to delete the child node just created. So we press F10 to access the menu, then use the keyboard to select the menu item Edit⇒Node⇒Remove Node. But now we observe a bug: instead of the child node being deleted, we see that a sibling node was created instead. So now the root has two children.

At this point, with the cursor still in the edit field of the newly created node, we press F11, a Stabilizer keyboard shortcut for reporting a bug from within the target application. When we do so, the Stabilizer pops up a "Report Bug" dialog. (See Figure 2.) When reporting a bug, we provide a *text description* by explaining what happened in words and we also provide a *visual description* by zooming in on the relevant parts of the before and after screenshots. (See Figure 2.) Although before and after screenshots of the entire screen are taken automatically by the Stabilizer, the user can additionally manually zoom in on the interesting part of each screenshot using the mouse.

After pressing Okay button on the "Report Bug" dialog, we immediately get a warning dialog giving us the option of aborting a FreeMind action. The warning dialog shows us past bug reports (and also "not bug" reports as we shall see shortly) ordered by increasing distance to the current context on which the bug prediction is made. (See Figure 3 for another warning dialog that will appear later in this example.)

This bug warning (not shown in figures) is based on one bug report, namely the one we just made. The warning stems from a "focus lost" event from the child node edit field. The reason this warning is given is that bug prediction takes into account past events by default. In this case, the bug report we just made shares some event history with the "focus lost" event.

We decide to allow the "focus lost" event to be processed, so we click Continue Action on the warning dialog. Indeed, after allowing execution to continue, we encounter no bug. So we click on the Report Not Bug button in the



Help

Monitoring:

Report Bug — freemind.main.FreeMind

Report Not Bug

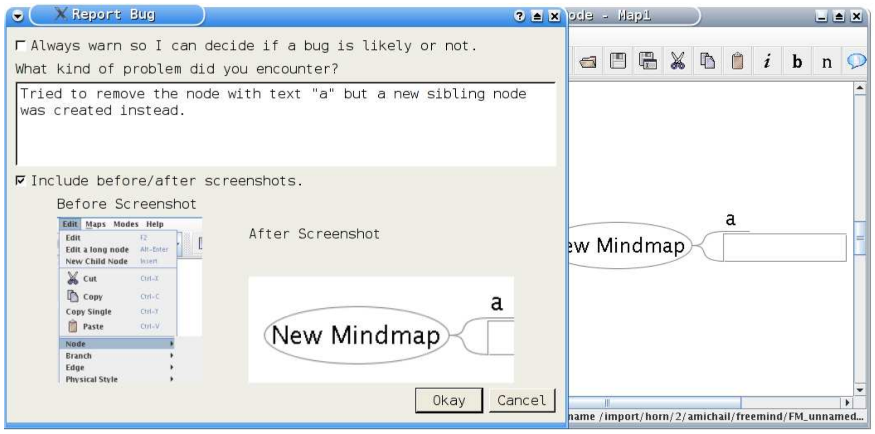
☑ Issue warnings on possibly buggy acti

Not Bug Score:

Distance	Report Filed	Before Screensh

Changing the zoom to the user defined zoom value of 200.0%

Figure 1. Running FreeMind under the Stabilizer.



☐ Always warn so I can decide if a bug is likely or not.

What kind of problem did you encounter?

Tried to remove the node with text "a" but a new sibling node was created instead.

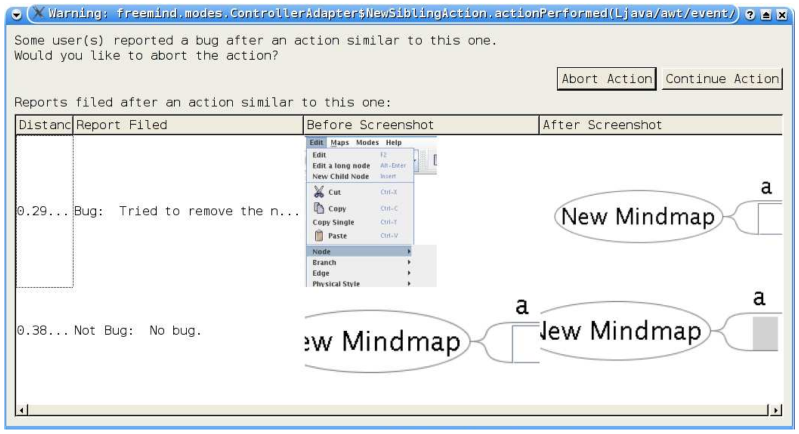
☑ Include before/after screenshots.

Before Screenshot

After Screenshot

name /import/horn/2/amichail/freemind/FM_unnamed...

Figure 2. Reporting a bug.



Warning: freemind.modes.ControllerAdapter$NewSiblingAction.actionPerformed(Ljava/awt/event/

Some user(s) reported a bug after an action similar to this one.
Would you like to abort the action?

Reports filed after an action similar to this one:

Distanc	Report Filed	Before Screenshot	After Screenshot
0.29...	Bug: Tried to remove the n...		
0.38...	Not Bug: No bug.		

Figure 3. A bug warning.

client GUI to indicate that the previous warning was in error – thus improving the Stabilizer's bug prediction. (The "Report Not Bug" dialog is similar to the "Report Bug" dialog shown in Figure 2.)

Now, we still wish to delete the child node as well as the new one that was created due to a bug. So we try to do it in a different way by using FreeMind's popup menu rather than the menubar at the top. Perhaps the node deletion bug may not occur. (In a real-life scenario with many users running FreeMind under the Stabilizer, it would be natural for some of those people to use the popup menu to delete a node.) Indeed, clicking the right mouse button for the popup menu (rather than F10 for the menubar at the top) and selecting the menu item Node⇒Remove Node does indeed delete the child as expected. So we delete both children in this way, leaving only the root node.

Now, we try to create a child node again for the root. Creating a child using the menubar at the top was not a problem before, so we try it again by pressing F10 to access the menubar, but doing so results in a warning. This is not surprising since F10 is in the history of the bug report that we made earlier. (If another user received this warning — unaware of our actions, "bug", and "no bug" reports — he/she could look at the warning dialog's list of "bug" and "no bug" reports, which are sorted by increasing distance to the current situation.) At this point, we click on Continue Action to ask the Stabilizer to process the F10 event and allow us to access the menu. As no bug subsequently occurs, we file a "not bug" report.

Next, we press F10 again and indeed there is no bug warning this time. We then select the menu item Edit⇒New Child Node. At this point, we get another warning, this time about the "new child" event. Again, this occurs because this event was in the history of the bug report made earlier. Since "new child" worked for us before, we would expect it to work again, so we click Continue Action and indeed a child is created with no bug. So we file a "not bug" report.

Now, we type the text "b" in the new child and press return. We get a warning on the return event. We ignore the warning and click Continue Action. No bug subsequently occurs and we file a "not bug" report.

At this point, we try to delete this child (with text "b") using the menubar via F10. Recall that this led earlier to a bug where a child is added instead. So we press F10, but no warning occurs. We then use use the keyboard to select Edit⇒Node⇒Remove Node. We now get a warning as expected. (See Figure 3. Observe that this warning includes both "bug" and "not bug" reports. The "bug" report has a lower distance to the current context as expected. Incidentally, the "not bug" report refers to the one we filed after the "focus lost" warning.) Inspecting the bug report in the warning dialog, we see that it looks very similar to the current situation, so we click Abort Action, thus avoiding the bug. We then use the popup menu to delete the child. We get no warning and the child is correctly deleted.

It turns out that the bug in question actually affects all menu items under Edit⇒Node in the menubar: if you access any of those menu items using the keyboard (e.g., starting with F10), a new sibling node will be created. So indeed, it makes sense to give warnings when F10 is followed by a "new sibling" event. The reason is this: even though there does exist a menu item Edit⇒Node⇒New Sibling Node, there are many other menu items under under Edit⇒Node as well, so if we see F10 followed by a "new sibling" event, the user probably picked some menu item other than Edit⇒Node⇒New Sibling Node, in which case there will be a bug. So a warning makes sense in this context as it is likely that a bug would occur.

3 Bug Prediction

The Stabilizer's bug prediction problem is this: given an application state S and an event e, would processing event e in state S likely result in a bug given past bug and "not bug" reports?

We shall use a bounded execution history to approximate the current application state S, which we describe in Section 3.1. The way in which the bounded history is used for bug prediction is explained in Section 3.2. The way in which the bug and "not bug" reports are used to provide training data for bug prediction is explained in Section 3.3. The actual bug prediction itself uses a distance weighted nearest neighbor learner, which is described in Section 3.4. The distance measure used for the learner is presented in Section 3.5. Finally, we describe support for manual bug prediction for those situations where learning is not working well (e.g., when our approximation to the state S is not good enough) in Section 3.6 .

Please note that the Stabilizer prototype intercepts (and possibly aborts) event callbacks. However, to simplify the exposition, sometimes we will not distinguish between events and their callbacks.

3.1 Bounded History

There are two obvious ways in which execution history could improve bug prediction. First, if a sequence of two or more actions is required to trigger a bug, then a history (provided it is long enough) would allow us to predict a bug when the sequence of those two or more actions is observed. Second, history can help us even when only one user initiated action is sufficient to trigger a bug. In particular, a user initiated action may actually result not only in one event (e.g., "new sibling" in FreeMind) but an "event burst" (e.g., "new sibling" quickly followed by a "focus gained" event for the edit field in the new sibling node). If the user subsequently reports a bug, the most recent event will not in fact be the one that initiated the action. History allows us to take into account previous events and give warnings on earlier events that initiate actions (e.g., "new sibling" rather than the subsequent "focus gained").

We keep track of two separate bounded histories: one for events and another for code (either function calls or basic blocks). Both histories are handled in the same way as described below. (The size of the event history is 10 by default, while the size of the code history is 100 by default.)

Let $H = (h_1, \ldots, h_n)$ denote a bounded history of size n. We shall define the history H in a way so that no item occurs multiple times in H, but at the same time, some sense of sequence is still preserved. In this way, more frequent items (e.g., more frequent events or function calls) will not dominate the bounded history. We shall use the notation $H + x$ to denote the addition of a new item x to $H = (h_1, \ldots, h_n)$. If item x is not already in H, then the first item h_1 in H is removed and item x is appended to the end: $H + x = (h_2, \ldots, h_n, x)$. If item x is already in H, say $H = (\ldots, h_i, x, h_j \ldots)$, then it is simply moved to the end: $H + x = (\ldots, h_i, h_j, \ldots, x)$.

Note that a sense of sequence is preserved in the history $H = (h_1, \ldots, h_n)$. In particular, suppose h_i precedes h_j in $H = (\ldots, h_i, \ldots, h_j, \ldots)$. Then it must be the case that the latest addition of h_j was done after the latest addition of h_i. Note also that if we add distinct items $x_1, \ldots, x_k$ in that order to $H = (h_1, \ldots, h_n)$ where $k \leq n$, then $x_1, \ldots, x_k$ will be the most recent items in the history (irrespective of whether some of those items were already in the history) while also retaining their order: $(\cdots((H + x_1) + x_2) + \cdots + x_k) = (\ldots, x_1, x_2, \ldots, x_k)$.

3.2 Validating Events

Let H_e and H_c denote the current event and code histories, respectively. H_e consists of all accepted events (i.e., those for which either no warning occurred or a warning did occur yet the user continued) and H_c contains either function calls or basic blocks leading up to the most recent event in H_e. Any more recent function calls/basic blocks are temporarily kept elsewhere and will be added to H_c later when an event following them is accepted and added to H_e.

Whenever an event x in the target application is about to be processed, its callback is intercepted by the Stabilizer runner. The runner then issues a "validate x" request to the Stabilizer client to determine whether it should proceed with or abort event x's callback. Whenever the Stabilizer client receives the "validate x" request, it performs a prediction based on the event and code histories, namely $(H_e + x, (\cdots((H_c + y_1) + y_2) + \cdots + y_k))$ where $y_1, y_2, \ldots, y_k$ denote function calls/basic blocks not yet added to H_c that precede event x, to determine whether a bug is likely to occur based on past data. (See Section 3.4.) Note that x is not an accepted event at this point, so it is not already part of the history H_e. We shall use (H_e^p, H_c^p) to denote the histories used for prediction, namely $(H_e + x, (\cdots((H_c + y_1) + y_2) + \cdots + y_k))$.

If the client predicts that a bug is likely, then a warning dialog will appear allowing the user to either abort or continue the action. If the user chooses to abort the action, the client will inform the Stabilizer runner to abort the current action and event x's callback will not be executed. Now if the client predicts that a bug is not likely or it predicts that a bug is likely but the user continues the action anyway, then the client will perform the same processing in both cases. Namely, the client updates the histories as follows: $H_e := H_e^p$, $H_c := H_c^p$. Now that x's callback has been accepted, it is now the most recent event in H_e and any calls/basic blocks leading up to x have been added to H_c.

3.3 Bug and "Not Bug" Reports

If the user reports a bug, then a training example consisting of the current event and code histories, H_e and H_c respectively, is added to the training data. Specifically, $(H_e, H_c, \text{"bug"})$ is added to the training set, where "bug" is the classification of this training example. Note that H_c contains only the code history leading up to the most recent event (since at bug prediction time we would not have the code that will execute after the event).

If the client predicts that a bug is likely and issues a bug warning, then the user continues the action anyway despite the bug warning, and yet the action turns out to be apparently bug-free, then the user can report "not bug" to tell the Stabilizer that the warning should not have occurred in this context. In this case, a training example consisting of the event and code histories used for prediction at the time of the previous warning, which we denote as $H_e^{p,w}$ and $H_c^{p,w}$, are added to the training data. Specifically, $(H_e^{p,w}, H_c^{p,w}, \text{"not bug"})$ is added to the training set, where "not bug" is the classification of this training example.

3.4 Distance Weighted Nearest Neighbor Learner

Bug prediction is done using the well-known distance weighted nearest neighbor learner [9, pp. 233–234]. The idea is to consider the "closest" k training examples to see whether a bug is likely or not, for some constant $k \geq 1$. More precisely, we use a distance measure $0 \leq d((H_e^p, H_c^p), (H_e', H_c') \leq 1$ to determine how close each training example $(H_e', H_c', \text{type})$ is to the event and code histories used for prediction, (H_e^p, H_c^p). (See Section 3.5 for our definition of d.)

Distance weighted nearest neighbor prediction is done as follows. If the distance to the closest training example is 0, then we take a majority vote on the classification among only those training examples of distance 0 to (H_e^p, H_c^p). So if the majority say there is a bug, then we predict a bug. Otherwise, we predict no bug.

If the distance to the closet training example is greater than 0, then we consider the closest k training examples to (H_e^p, H_c^p), say $(H'_{e,1}, H'_{c,1}, \text{type}_1), \ldots, (H'_{e,k}, H'_{c,k}, \text{type}_k)$, where we exclude training examples of maximum distance 1. (The number of such training examples may be less than k if there is insufficient training data. Also, by default, the Stabilizer actually considers all neighbors with distance less than 1, which is reasonable as we shall take into account the distance to each of those neighbors when making a prediction. However, users can if they wish specify a k value to restrict the numbers of neighbors considered to only the closest k ones with distance < 1.)

We then compute two scores, one for the classification "bug" and the other for the classification "not bug". Let X denote the set of training examples among $(H'_{e,1}, H'_{c,1}, \text{type}_1), \ldots, (H'_{e,k}, H'_{c,k}, \text{type}_k)$ where type$_i$ is "bug". Let Y denote the set of training examples among $(H'_{e,1}, H'_{c,1}, \text{type}_1), \ldots, (H'_{e,k}, H'_{c,k}, \text{type}_k)$ where type$_i$ is "not bug". The "bug" score is computed by $\sum_{(H'_e, H'_c, \text{"bug"}) \in X} 1/d((H_e^p, H_c^p), (H'_e, H'_c))^2$ and the "not bug" score is computed by $\sum_{(H'_e, H'_c, \text{"not bug"}) \in Y} 1/d((H_e^p, H_c^p), (H'_e, H'_c))^2$. If the "bug" score is greater, we predict a bug, otherwise we predict a "not bug".

3.5 Distance Measure used in Learner

As before, we let H_e^p and H_c^p denote the event and code histories used for prediction and we let H'_e and H'_c denote a training example event and code histories. Since our bounded histories do not contain duplicate items, we shall at times treat histories as sets, as there will be no confusion given the context.

We compute a normalized distance between the prediction event and code histories and the training example event and code histories, which we denote by $0 \le d((H_e^p, H_c^p), (H'_e, H'_c) \le 1$. This is done as follows. If the event added most recently to H_e^p (i.e., the last in the sequence) is not present in H'_e, then $d((H_e^p, H_c^p), (H'_e, H'_c) = 1$ and the procedure for determining distance is completed at this point. The rationale here is that a bug warning if any would be about the most recent event and so this event is absolutely critical to bug prediction. Thus, we require that the most recent event be present in H'_e to give a distance $d((H_e^p, H_c^p), (H'_e, H'_c)$ that is less than one.

If the event added most recently to H_e^p (i.e., the last in the sequence) is present in H'_e, then we compute the standard cosine similarity from information retrieval [11, p. 185] of H_e^p and H'_e:

$$S_e(H_e^p, H'_e) = \frac{\sum_{x \in H_e^p \cap H'_e} w_e^p(x) w'_e(x)}{\sqrt{\sum_{x \in H_e^p} w_e^p(x)^2} \sqrt{\sum_{x \in H'_e} w'_e(x)^2}}$$

where the weight of an item x, denoted by $w_e^p(x)$ for H_e^p and $w'_e(x)$ for H'_e, is set to r_e^{i-1}, where x is the i^{th} item in the corresponding history and the ratio $r_e \ge 1$ is a constant. (Values of $r_e > 1$ give greater weight to more recent items in the histories. We use $r_e = 1.5$ by default.)

We also compute the cosine similarity of the code histories, H_c^p and H'_c:

$$S_c(H_c^p, H'_c) = \frac{\sum_{x \in H_c^p \cap H'_c} w_c^p(x) w'_c(x)}{\sqrt{\sum_{x \in H_c^p} w_c^p(x)^2} \sqrt{\sum_{x \in H'_c} w'_c(x)^2}}$$

where the weight of an item x, denoted by $w_c^p(x)$ for H_c^p and w'_c for H'_c, is set to r_c^{i-1}, where x is the i^{th} item in the corresponding history and ratio $r_c \ge 1$ is a constant. (We use $r_c = 1.1$ by default.)

We then compute the combined similarity $0 \le S((H_e^p, H_c^p), (H'_e, H'_c) \le 1$ as follows:

$$S((H_e^p, H_c^p), (H'_e, H'_c) = \alpha S(H_e^p, H'_e) + (1-\alpha) S(H_c^p, H'_c),$$

for some constant $0 \le \alpha \le 1$ that is used to determine how important the event history similarity score is with respect to the code history similarity score. (We used $\alpha = 1$ for the motivating example in Section 2, which means that the code histories were ignored. When we do use both event and code histories, we typically use $\alpha = 0.5$, as is done in the evaluation configurations that make use of code history in Section 4.)

Finally, the distance d is defined as follows: $d((H_e^p, H_c^p), (H'_e, H'_c) = 1 - S((H_e^p, H_c^p), (H'_e, H'_c)$.

3.6 Support for Manual Bug Prediction

In situations when learning is not effective (e.g., because our approximation to the state is insufficient), the user can override automated bug prediction. In particular, when filing a bug report, the user can specify "always warn". Moreover, when filing a "not bug" report, the user can specify "never warn". When looking at neighbors, if among the closest ones there is a neighbor with "always warn", then a warning will be given regardless of the other neighbors. Otherwise, if among the closest ones there is a neighbor with "never warn", then a warning will not be given regardless of the other neighbors. This feature should only be used as a last resort when learning is ineffective in a particular context. (It is not used in the motivating example in Section 2 nor is it used in the evaluation in Section 4.)

4 Evaluation of Bug Prediction

This section presents the preliminary experiment conducted to evaluate the Stabilizer's capability of automated bug prediction. We first describe the experimental subjects, including the subject programs, mutants, and tests. We then discuss the experiment design. We finally present the experimental results and discuss threats to validity.

4.1 Subjects

The subjects used in the experiment had been previously developed and used by Memon et al. in evaluating different types of GUI test oracles [8]. The subject programs include four GUI applications. The first three columns of Table 1 show the program names, the number of lines of code, and the number of classes, respectively. For each application, Memon et al. created 100 mutants (i.e., 100 versions each of which is seeded with a bug) based on the collected bugs introduced during the development of these four applications. They used an automated tool to generate 600 test cases for each application (independently of the mutants). They also developed a GUI test replayer for running these tests automatically.

A *mutated method* is a mutant's method that is seeded with a bug. To automatically determine whether an execution exposes a bug in a mutated method, we manually inspect the original code and mutated code, and derive the bug-exposure condition under which the execution of the mutated code can cause differences in the program state or return value at the exit of the mutated method. At an appropriate location within the mutated method, we manually insert a piece of code that informs the runner when the bug-exposure condition is satisfied.

A *bug-triggering callback* for a mutant is a callback that at least once directly or indirectly invoked the mutated method during the executions of the 600 tests. Note that one particular execution of a bug-triggering callback does not necessarily expose a bug or even cover the mutated method. To control the scale of the experiment, for each mutant, we select all those tests that execute at least one triggering callback for the mutant (no matter whether these tests expose bugs) and use the Stabilizer to run the selected tests on the mutant. Because unselected tests for a mutant exercise application features irrelevant to the buggy code, we expect that our experimental results shall be similar to the ones of running all 600 tests for each mutant.

A bug is *deterministic* with respect to a bug-triggering callback c if whenever c is executed, the bug is guaranteed to be exposed within the execution of c, and is *nondeterministic* with respect to c if the bug is exposed by at least one but not all executions of c among the executions of 600 tests. Because deterministic bugs are trivial for the Stabilizer to predict, we select a mutant in the experiment if the mutant contains a nondeterministic bug with respect to one of its bug-triggering callback. However, we do not select a mutant if the number of selected tests for the mutant is fewer than 20 because the Stabilizer assumes a repeated use of a bug-triggering callback; we do not select a mutant with all 600 selected tests because the mutant might contain a bug that is not related to event callbacks. The fourth and fifth columns of Table 1 show the number of selected mutants and the average number of selected tests for a selected mutant. The third application, TerpPaint, has zero selected mutants because none of its mutants satisfy our selection criteria.

program	loc	classes	mutants	tests
TerpWord	1747	9	2	170
TerpPresent	4769	4	9	56
TerpPaint	9287	42	0	–
TerpSheet	9964	25	8	150

Table 1. Subject programs used in the experiment

4.2 Experiment Design

The objectives of the experiment are to investigate the following research questions:

- **RQ1:** Can event history or code history (i.e. regular method calls or basic blocks) be useful in improving the automated bug prediction?
- **RQ2:** Can lower-level execution information be useful in improving the automated bug prediction (i.e. the arguments of event callbacks or the arguments/returns of regular method calls [1])?
- **RQ3:** Can the Stabilizer's automated bug prediction be improved over time?

To answer these questions, we at first define a default configuration to be using only events and event callback arguments in prediction and setting the event history size as 10. We then design six configurations for the experiment:

- **Configuration 1:** default configuration excluding event callback arguments.
- **Configuration 2:** default configuration.
- **Configuration 3:** default configuration with the addition of regular method calls.
- **Configuration 4:** default configuration with the addition of regular method calls with argument/return values.

[1]We collect the label and the type of the component associated with an event as the argument of the event's callback. For example, when a user selects an $Edit$ menu, the callback argument is collected as "$Edit\#javax.swing.JMenu\#$". For a method argument or return, besides its runtime-type name, we collect its value in a string form if it is a primitive type, collect "null" if it is a null reference, and collect "not null" if it is a non-null reference.

- **Configuration 5:** default configuration with the addition of basic blocks.
- **Configuration 6:** default configuration with the event history size set to one (i.e., only the most recent event is used for prediction).

The code history size is set to be 100 for Configurations 3, 4, and 5, where code information is used.

To characterize the effectiveness of automated bug prediction, we use two measures: precision and recall, which are standard measures from information retrieval [11]. In our context, *precision* is defined to be the number of correctly predicted buggy events divided by the total number of bug warnings. *Recall* is defined to be the number of correctly predicted buggy events divided by the total number of events that were actually buggy. Note that these two measures do not involve the total number of events encountered, buggy or not. For each combination of selected mutants and six configurations, we calculate precision and recall over the execution of all selected tests for the selected mutant. Note that the execution of each combination starts with an empty Stabilizer database. In addition, to answer the last question (RQ3), for each combination we divide the sequence of executed callbacks into four parts (called periods) of equal size. Then we calculate precision and recall for each period by considering all those callbacks that are executed before and within the period.

Normally, whenever the Stabilizer client is asked to validate an event, it sends the runner an "abort" or "proceed" to tell it whether the event callback should be aborted or executed. However, "abort" is only sent after a bug warning in which the user has decided to abort the action. "proceed" is sent either if there is no bug warning, or there is a bug warning and the user has decided to continue the action. However, to build an automated evaluation setup, we need to take human users out of the loop. Consequently, we need to simulate the user's behavior on a bug warning dialog. For the purposes of automated evaluation, whenever the client is asked to validate an event callback c, instead of popping up a bug warning dialog, it simply continues the action (thus corresponding to the case where a human user would ignore the warning and continue the action). During the execution of the action, if no bug-exposure condition is satisfied, a "not bug" is automatically reported; if a bug-exposure condition is satisfied, a "report bug" is automatically reported.

4.3 Experimental Results

Figures 4-7 use boxplots to present the experimental results. The box in a boxplot shows the median value as the central line, and the first and third quartiles as the lower and upper edges of the box. The whiskers shown above and below the boxes technically represent the largest and smallest observations that are less than 1.5 box lengths from the end of the box. In practice, these observations are the lowest and highest values that are likely to be observed.

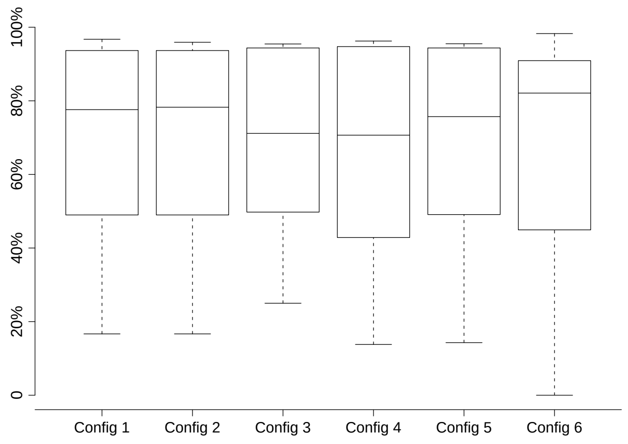


Figure 4. Precision for the six configurations.

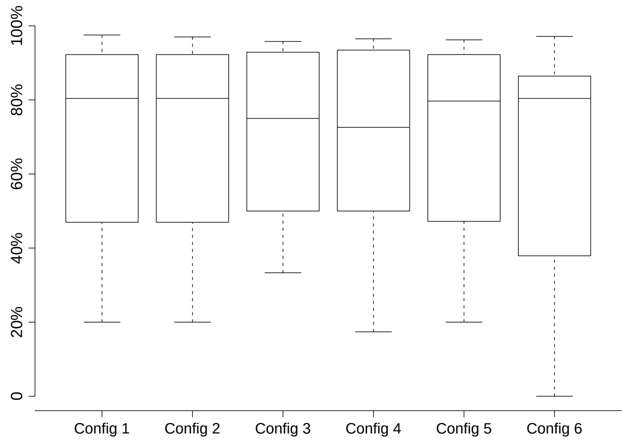


Figure 5. Recall for the six configurations.

Figures 4 and 5 show the precision and recall for the mutants with the six configurations. Figures 6 and 7 show the precision and recall for the mutants over the four periods with the combined data from all six configurations. From Figures 6 and 7, we have observed that precision and recall were increased over time, indicating that the Stabilizer had learned over time. From Figures 4 and 5, we have observed that the Stabilizer can achieve around an 80% median for precision and recall. But it seems it would be hard to improve since we tried some obvious lower-level execution information (i.e., regular method call history, basic block history, callback argument values, regular call argument/return values) and did not get discernable improvement. However, it is clear to us that event history is important, at least to handle event bursts (e.g., in FreeMind); this issue was discussed in Section 3.1. We speculate that if we want to improve significantly on the Stabilizer's bug prediction performance, we would need to look at much more of the program state.

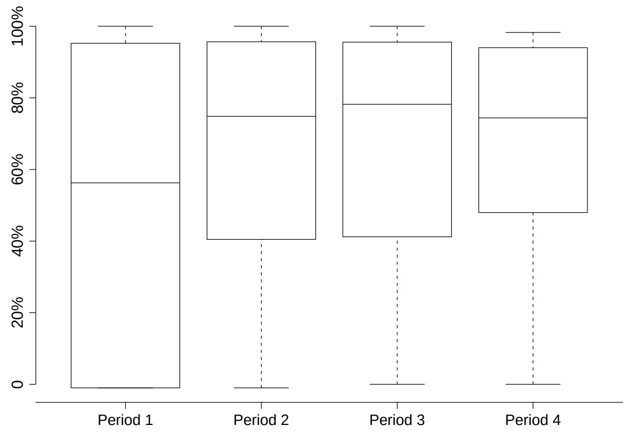


Figure 6. Precision for the four periods.

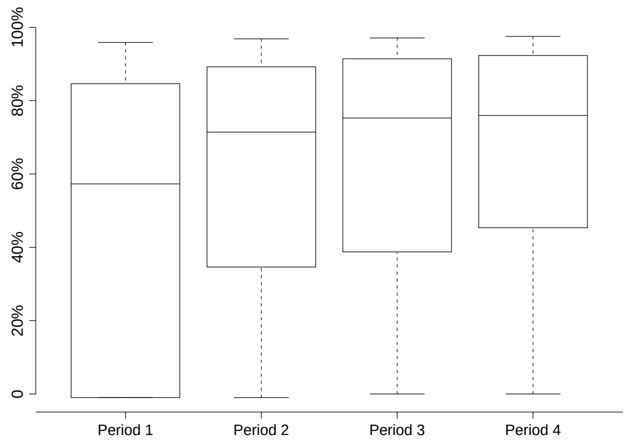


Figure 7. Recall for the four periods.

4.4 Threats to Validity

The threats to external validity primarily include the degree to which the subject programs, bugs, and test cases are representative of true practice. Although the GUI applications used in the experiment are fairly large with complex GUIs, they do not reflect a wide spectrum of possible GUI applications. In addition, the total number of selected mutants in the experiments is only 19. These threats could be reduced by more experiments on wider types of subjects in future work. The threats to internal validity are instrumentation effects that can bias our results. Faults in our prototype and the test replayer might cause such effects. To reduce these threats, we manually inspected many collected traces for each subject. In addition, the selection of mutants and their tests in the experiment might also cause instrumentation effects. Threats to construct validity arise when manually derived bug-exposure conditions might not reflect the actual bug-exposure conditions when buggy code is used by real users. In the future, we plan to conduct case studies on the Stabilizer being used by real users.

5 Related Work

The cooperative bug isolation project developed by Liblit et al. [7] collects predicate and crash information about deployed software from a set of runs produced by users. They develop the predicate elimination and statistical debugging techniques to identify a predicate as a bug-exposure condition for a crash-causing bug. Our approach operates at deployment sites too, but focuses on a narrower range of applications: GUI applications. Our approach considers any undesirable behavior as a bug, whereas the cooperative bug isolation project is mainly concerned with program crashes.

Comparing to bug isolation in general, the final results (prediction results) of our approach are directly used by application users, whereas the final results (fault-exposure conditions) of bug isolation are mainly used by developers (the cooperative bug isolation project deliberately makes its approach to be unobtrusive to user behavior). Data available for predicting a buggy event callback are limited to the execution information before a buggy event callback, whereas data available for finding bug-exposure conditions in bug isolation are all the execution information before the bug-exposure site (even after the application has started the execution of the buggy event callback). In this regard, we speculate that the bug prediction problem in our particular setting might be more challenging than the bug isolation problem in general. On the other hand, bug isolation needs to produce human-understandable bug-exposure conditions, which could be used but not required in bug prediction; in this regard, bug isolation is more challenging.

Given a faulty run and a larger number of correct runs, Manos and Reiss [10] select the correct run whose basic block profiling is closest to the faulty run. They then compare the structural spectra of the selected correct run and the buggy run, and report the suspicious parts of the program as buggy portions. Given an event callback to be executed and a set of historical runs of the same callback, our approach predicts whether the callback to be executed is buggy using the nearest-neighbor strategy.

Given a failing test, Zeller and Hildebrandt [12] develop the Delta Debugging algorithm to systematically generate and run test inputs that are slightly different from the failing test input, in order to isolate the parts in the input that cause the failure. Different from their approach, our approach does not proactively generate tests to exercise the callback to be predicted but takes advantage of historical execution information exercised by users. Our approach provides mechanisms to share the collective knowledge of user executions among various sites alleviating the problem of lacking enough data for prediction.

Elbaum et al. [4] empirically investigate the relationship between anomalies and failures by evaluating the predictive capabilities of various anomaly detection schemes in the context of failure analysis. They use behaviors exposed by in-house integration testing to define normal behaviors. Then they detect anomalies by detecting deviations from normal behaviors. We can view characteristics of a callback's passing runs as normal behaviors and characteristics of its failing runs as abnormal behaviors. Given a program behavior, Elbaum et al.'s approach detects its deviations from normal behaviors, whereas our approach determines whether it is more similar to historical normal behaviors (passing runs) or abnormal behaviors (failing runs).

Demsky and Rinard [3] develop an approach for dynamically detecting and repairing data structures that violate specified consistency constraints, rather than attempting to increase the reliability of the code manipulating the data structures. Their approach enables a program to continue to run successfully within its designed operating envelope. Our approach takes a similar perspective on making buggy code usable: we dynamically predict and prevent the execution of buggy code before it gets fixed. However, without requiring any specification, we attempt to prevent a GUI application from entering a corrupted state instead of aggressively repairing an already-corrupted state.

Our approach is related to intrusion detection research in computer security. In our case, we attempt to avoid bugs, rather than intrusions. However, our use of a supervised learner (nearest neighbor) on a bounded execution history is similar to the sliding window nearest neighbor method used in intrusion detection systems [6].

Finally, our use of before/after screenshots to visually describe application state at a very high level of abstraction is similar to work done on editable graphical histories [5] and the DRT design recovery tool for interactive graphical applications [1].

6 Conclusions

In this paper, we have proposed a tool-based approach to help users avoid bugs in GUI applications. The idea is that users would use the application normally and report bugs (and also "not bugs") that they encounter to prevent anyone — including themselves — from encountering those bugs again. When a user attempts an action that has led to problems in the past, he/she will receive a warning and will be given the opportunity to abort the action — thus avoiding the bug altogether and keeping the application stable. We have illustrated our approach by example using our Stabilizer prototype, explained how the Stabilizer's bug prediction works, and presented a preliminary evaluation of the Stabilizer's bug prediction.

For future work, we would like to test the Stabilizer with multiple users. The Stabilizer prototype was designed with distributed operation in mind, so implementation-wise, supporting multiple users on different computers will be easy and we plan to do this soon. However, evaluation of Stabilizer usage with multiple users will require some thought. In particular, we would like to get some sense of how to predict the "stabilization time" of an application given a host of factors: the number of users, the number of bugs, the size of the application, etc. Also for future work, we would like to address privacy (e.g., users inadvertently sending sensitive information in a bug or "not bug" report) and security (e.g., competitors crippling an application through dishonest bug and "not bug" reports).

Acknowledgments

We would like to thank Atif Memon and Qing Xie for their assistance in our use of the TerpOffice benchmarks.

References

[1] K. Chan, Z. Liang, and A. Michail. Design recovery of interactive graphical applications. In *Proceedings of the 25th International Conference on Software Engineering*, pages 114–124, 2003.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proc. 18th ACM Symposium on Operating Systems Principles*, pages 73–88, 2001.

[3] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Proc. 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 78–95, 2003.

[4] S. Elbaum, S.Kanduri, , and A. Andrews. Anomalies as precursors of field failures. In *Proc. International Symposium of Software Reliability Engineering*, pages 108–118, 2003.

[5] D. Kurlander and S. Feiner. Editable graphical histories. In *IEEE Workshop on Visual Languages*, pages 127–134, 1988.

[6] T. Lane and C. E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Information and System Security*, 2(3):295–331, 1999.

[7] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 141–154, 2003.

[8] A. M. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 164–173, 2003.

[9] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[10] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 30–39, 2003.

[11] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[12] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.