CodeDeviant: Helping Programmers Detect Edits That Accidentally Alter Program Behavior

Austin Z. Henley
Department of Electrical Engineering & Computer Science
University of Tennessee
Knoxville, Tennessee 37996-2250
azh@utk.edu

Scott D. Fleming
Department of Computer Science
University of Memphis
Memphis, Tennessee 38152-3240
Scott.Fleming@memphis.edu

Abstract—**In this paper, we present CodeDeviant, a novel tool for visual dataflow programming environments that assists programmers by helping them ensure that their code-restructuring changes did not accidentally alter the behavior of the application. CodeDeviant aims to integrate seamlessly into a programmer's workflow, requiring little or no additional effort or planning. Key features of CodeDeviant include transparently recording program execution data, enabling programmers to efficiently compare program outputs, and allowing only apt comparisons between executions. We report a formative qualitative-shadowing study of LabVIEW programmers, which motivated CodeDeviant's design, revealing that the programmers had considerable difficulty determining whether code changes they made resulted in unintended program behavior. To evaluate CodeDeviant, we implemented a prototype CodeDeviant extension for LabVIEW and used it to conduct a laboratory user study. Key results included that programmers using CodeDeviant discovered behavior-altering changes more accurately and in less time than programmers using standard LabVIEW.**

I. Introduction

A particularly difficult activity for programmers is understanding how their changes to code affect other parts of the program. Because software is made up of many interrelated code modules, a small change in one module can have cascading effects throughout the rest of the program. Moreover, code is often missing explicit information about the relationships between code modules (known as *hidden dependencies* [15]). To understand the full impact of a code change, programmers must possess a correct mental model of the source code. In fact, researchers have generally found that people require a rich mental model before they can organize information on an even less complex task, such as choosing the best camera to purchase [20]. However, forming such mental models about programs is a notoriously error-prone and time-consuming task due to the sheer size and complexity of modern software [33]. This is further complicated by the fact that programmers' information needs are rapidly changing as they work through programming tasks [34], [35].

In this paper, we focus on a large class of code changes, known as *refactorings*, specifically in the context of visual programming environments. Refactoring aims to improve the design of a program by changing the structure of its code without altering its behavior [12], [31]. It has become a ubiquitous practice in software development [19], [28]. Surveys of programmers have indicated that programmers find refactoring to be an important part of the development process [7], and they believe it provides a variety of benefits, including improving readability and extensibility [19]. Studies of both textual and visual programming languages have provided some support for these views, empirically demonstrating the benefits refactoring can provide. In particular, studies of textual languages have shown that refactoring improved maintainability [21] and reusability [27] of code. Although little work has been done to study refactoring in visual languages, one study did find that programmers preferred code that had been refactored to remove code smells [43].

Despite refactoring's popularity and benefits, it is often difficult for programmers, both of textual and visual languages, to perform. Most programming environments for textual languages provide features for automated refactorings, but programmers rarely use them [13], [19], [28], [30], [44]. Several reasons have been cited for the underutilization of such features. The tools are not trusted by programmers [44]; they provide unhelpful error messages [28]; and they have even been found to introduce bugs [4], [10], [39], [41], [44], [46]. However, refactoring manually is a tedious process that has also been found to be error prone in textual languages [13] as well as in visual languages [17].

A particularly difficult aspect of refactoring observed among textual-language programmers is ensuring that code changes did not alter the behavior of the program. Best practices suggest the use of test suites, which allow programmers to define correct program behavior and to be alerted whenever a code change causes a test to fail. Creating such software tests is a common approach to ensuring software quality in contemporary software development. However, refactorings may break the code that tests the software [24], [36], [39]. Moreover, software tests have also been found to be inadequate at finding refactoring errors [37], and having tests available during refactoring did not improve the quality of the refactorings produced by programmers [45]. To address these shortcomings of software testing, researchers have recently proposed tools to detect manual refactorings, automatically complete them, and validate their correctness (e.g., BeneFactor [13], GhostFactor [14], and WitchDoctor [11]). However, these tools do not support all types of refactorings.

To better understand the challenges of refactoring in visual languages, we conducted formative investigations of programmers of the visual dataflow language, LabVIEW, engaged in refactoring, and found that, similar to textual-language programmers, they also have considerable difficulties in attempting to validate their code changes. In particular, the programmers reported that they often introduce bugs unintentionally while refactoring. To cope with this issue, these programmers followed tedious strategies to detect behavior-altering changes, such as writing down program output on a piece of paper prior to the code change. An analysis of the programmers' refactorings found that not only were buggy refactorings common, but that they did indeed alter the program output unintentionally.

To address these issues of detecting behavior-altering changes, we propose CodeDeviant, a novel tool concept for visual programming language environments. The CodeDeviant tool compares the program output with a previous execution to determine whether the behavior has been altered. CodeDeviant aims to integrate seamlessly into programmers' workflows by not requiring any upfront effort (unlike creating software tests). Key features of CodeDeviant include transparently recording program execution data, enabling the programmer to efficiently compare program outputs, and allowing only apt comparisons between executions.

To evaluate the success of our CodeDeviant design, we implemented a prototype CodeDeviant extension for LabVIEW and conducted a laboratory study involving 12 professional LabVIEW programmers. The evaluation compared our CodeDeviant-extended LabVIEW with standard LabVIEW for two key criteria: how accurately the programmers could spot code changes that changed a program's execution behavior and how quickly the programmers could make such decisions.

This work makes the following contributions:

- The findings of formative investigations of programmers refactoring in LabVIEW showing, among other things, the difficulty that programmers had in verifying that their code edits did not alter program behavior.
- A novel tool design based on our formative findings, CodeDeviant, for efficiently detecting behavior-altering changes while integrating seamlessly into programmer workflows.
- A prototype of CodeDeviant implemented as an extension to the LabVIEW visual dataflow programming environment.
- The results of a lab evaluation of CodeDeviant showing that programmers more accurately and more quickly identified behavior-altering code changes with CodeDeviant-extended LabVIEW than with standard LabVIEW.

II. Background: Visual Dataflow Languages

In this paper, we focus on programmers refactoring visual dataflow code. Unlike textual programming languages (e.g., Java), visual dataflow code consists of boxes (functions) and wires (values). For example, Fig. 1 depicts a small visual dataflow program. Even though visual dataflow languages have a drastically different syntax than textual languages, they have many of the same features, such as modularity.

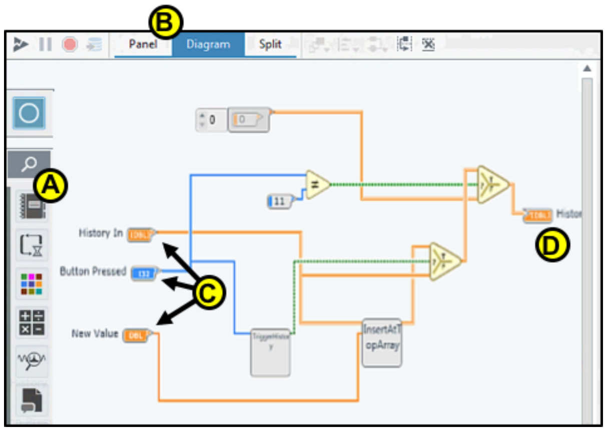


Fig. 1. LabVIEW block-diagram editor. Editors have (A) a palette, along with (B) debugging and other controls. The pictured editor has open a code module with (C) multiple inputs and (D) one output.

In particular, we focus on LabVIEW, a commercially available visual dataflow programming environment that is one of the most widely used visual programming languages [48]. LabVIEW programs are composed of code modules called *Virtual Instruments* (VIs). Fig. 1 illustrates the LabVIEW editor with a particular code module open. This VI has several inputs (e.g., Fig. 1-C) and an output (Fig. 1-D). This example consists of a series of operations performing some logical comparisons (yellow triangles) and calling two other VIs (gray boxes). An important characteristic of VIs is that they can run continuously, allowing multiple rounds of executions to be performed—that is, taking in different inputs and returning different outputs repeatedly before the execution is terminated.

III. Formative Investigations

To understand the problems that programmers have while refactoring, we performed two formative investigations. In the first, we qualitatively shadowed programmers as they performed refactorings for their jobs, and interviewed them based on our observations. Based on these findings, we began to explore possible solutions, and conducted a second investigation to test the feasibility of one of our candidate solutions.

A. Qualitative Shadowing Study

To better understand how LabVIEW programmers refactor code and the problems they encounter, we performed *qualitative shadowing* [23]. Following this method, a session involved sitting behind a programmer for at least one hour while they worked in their own environment. Our participants were 9 professionals that program every day using LabVIEW. Their job titles consisted of 4 systems engineers, 2 software engineers, 2 application engineers, and 1 hardware engineer. To initiate these shadowing sessions, we emailed the programmers, asking them to arrange a time for us to come to their desk to observe them whenever they planned on "refactoring or cleaning up" code. We observed them while refactoring code, and finished by having a semi-structured interview based on our observations. To elicit feedback on our observations and address follow-up questions, we presented our findings to the same programmers in a group setting.

While working, all nine of the programmers indicated that they have difficulties while refactoring. Based on their verbal remarks while working and their interview responses, a significant issue was that they often introduce bugs while refactoring. One programmer demonstrated an example of this issue: while dragging a few elements around, he accidentally detached a wire from a VI, which changed the program's behavior (but did not produce a compiler error). Another example we observed was when a programmer performed a refactoring, he mixed up two of the outputs, accidentally causing a behavior-altering change (again, without producing a compiler error).

These types of bugs could have potentially been caught by unit tests; however, none of these programmers typically write such tests. In fact, only 3 of the 9 programmers had ever used unit tests in LabVIEW before, and 4 others had only used them in other languages. They indicated that it is too much work to create the unit tests, especially when they do not intend on making changes to the code in the future. One programmer mentioned that if the project uses unit tests, he has to continuously keep the unit tests up to date, which requires too much time for his workflow.

To cope with the difficulties of validating their refactorings, the programmers utilized a variety of strategies. For instance, we observed a programmer using a piece of paper to write down the output of test cases of a VI before he rewrote it. Three other programmers made copies of their entire project prior to making their changes, such that they could run the original version and the modified version simultaneously, to test the behavior. Other times, programmers ran the application to see if the behavior changed, but relied on their recollection of how the program behaved. However, these strategies are error-prone and tedious for the programmer to perform.

B. Feasibility Study

To explore possible solutions for detecting behavior-altering bugs, we analyzed six videos of LabVIEW programmers refactoring code from a prior study [17]. Our goal was to see how often programmers performed refactorings that unintentionally altered the behavior of the program and if that behavior change could be identified by observing the program's output. In the original study, programmers were tasked with refactoring various portions of an existing calculator application.

Each session lasted approximately 90 minutes. First, participants received an introduction to the code base they would be working on. Then, they were instructed that for the next 60 minutes they would be refactoring the code. To help the programmers get started, they were provided with three specific refactorings to perform. Afterwards, they were tasked with continuing to refactor the code however they saw fit. If they got stuck, they were given suggestions of other refactorings. The last 30 minutes of the session involved playing back portions of the video to the participant, and asking questions about each refactoring that they performed.

To analyze whether refactorings altered the program output, we looked at each refactoring episode using the screen-recording video and the participants' talk-aloud data. We considered the refactoring to be buggy if it resulted in behavior that was different than before the change. We inspected the output values to see if the program output could be used to identify the behavior-altering change. For this analysis, we assumed that the programmer would have executed the program before and after the change using the same input.

TABLE I
THE REFACTORING EPISODES WE ANALYZED TO SEE IF PROGRAMMERS INTRODUCED BUGS AND IF THE OUTPUT WAS CHANGED. ALTHOUGH P5 DID INTRODUCE ONE BUG, HE DID NOT COMPLETE THE TASK SO IT WAS EXCLUDED FROM OUR ANALYSIS.

Participant	Total refactorings	Buggy refactorings	Behavior-altering
P1	4	2	100%
P2	7	4	100%
P3	5	2	100%
P4	10	2	100%
P5	5	0	--
P6	5	1	100%

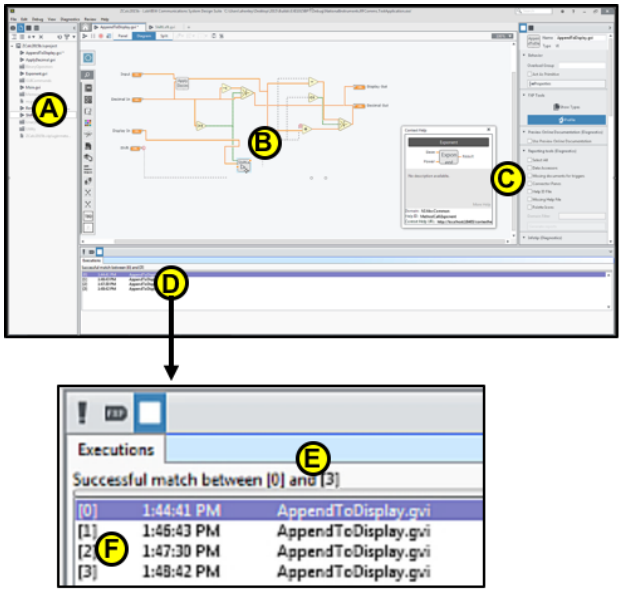
As shown in Table I, participants often introduced bugs while refactoring. In fact, all but one participant performed buggy refactorings, and on average, 31% of their refactorings were buggy. Furthermore, every bug they introduced caused the output of the program to change. A particularly common bug was wiring code elements incorrectly (e.g., swapping two wires). Other bugs included not handling corner cases for rewritten code (e.g., if the input is zero), changing a value in some locations but not all the locations, incorrectly initializing variables that were moved out of a loop or inner VI, and extracting a method but not calling it correctly. Understanding these behavior-altering changes were an initial step towards designing a tool that could detect these changes.

IV. Tool Design

To address the problems programmers have in detecting behavior changes after refactoring, we designed the CodeDeviant tool for visual dataflow programming environments. CodeDeviant enables programmers to compare the program output of a previous execution to the current execution so they can determine if their refactoring had unintended side effects on program behavior. By providing this information to programmers, CodeDeviant aims to enable programmers to test their changes more quickly and more accurately.

Based on our qualitative shadowing observations and feedback provided by programmers, we conceived of three key design principles for CodeDeviant:

- Transparently record program execution data as executions are performed by the programmer. In particular, CodeDeviant records the input values, output values, and metadata (e.g., timestamp) for each execution.
- Enable the programmer to efficiently compare selected program executions from the recorded history to see if the



Executions

Successful match between [0] and [3]

```
[0]   1:44:41 PM   AppendToDisplay.gvi
[1]   1:46:43 PM   AppendToDisplay.gvi
[2]   1:47:30 PM   AppendToDisplay.gvi
[3]   1:48:42 PM   AppendToDisplay.gvi
```

Fig. 2. CodeDeviant-extended LabVIEW IDE. The standard LabVIEW IDE features include (A) a project explorer, (B, see also Fig. 1) the code editor, and (C) contextual help and properties. CodeDeviant extends the IDE with (D) an additional pane that provides (F) a history of executions and that indicates (E) whether the behavior has changed.

program outputs changed while integrating seamlessly into their workflow.

- Only allow apt comparisons between executions and notify the user if there is not a suitable comparison. For example, if an input parameter is removed, it is not clear how CodeDeviant would compare the values.

To evaluate our design, we implemented CodeDeviant as an extension to the LabVIEW development environment. The remainder of this section explains the specific features of our CodeDeviant extension that satisfy these principles.

A. *Transparently Record Program Executions*

In our LabVIEW implementation, whenever a programmer executes a code module (i.e., a VI), CodeDeviant automatically records the sets of input and corresponding output values as well as metadata (e.g., timestamp and information about the VI). Recording is done transparently, not requiring any explicit user action. To implement this feature, we leveraged LabVIEW's compiler framework to walk the abstract syntax tree of the VI to identify the input and output nodes (recall Fig. 1-C,D). We then utilized LabVIEW's existing runtime framework, which already has highly optimized features for asynchronously logging high volumes of streaming data to a file. Being able to handle streaming data is an important criteria since LabVIEW is often used to stream vast amounts of data from instruments (e.g., an oscilloscope).

B. *Efficiently Compare Program Executions*

The main goal of CodeDeviant is to efficiently compare program executions to detect whether the behavior has changed. After an execution is finished, an entry is added to the history pane, depicted in Fig. 2-F, which displays the VI name and timestamp of when it was executed. To perform a comparison, the user selects which prior execution to use as a point of comparison by clicking the associated row in the history listing. Then, the next time the application is executed, the current execution will be compared to the selected execution.

To compare the executions, CodeDeviant inspects the logged input and output values. It first performs an intersection of the input values from both executions (and ignores any values that were not present in both executions). This step is needed because unlike functions in most textual languages, a VI can execute continuously, either streaming data from hardware or acting as a long-running interactive system. For this reason, VIs can be continuously fed inputs and continuously provide outputs—unlike in Java, for example, where a function call takes in one set of arguments and returns one value. CodeDeviant then compares whether the outputs are equivalent between executions, given the same inputs. For example, if execution A has two input/output pairs ((2,8), (5,20)) and execution B has three pairs ((3,12), (5,11), (8,8)), CodeDeviant would do an intersection on the inputs values of A and B, which in this case contains only the input value 5. The next step is to compare the corresponding outputs given 5 as input, which in this case, do not match ($20 \neq 11$).

Once CodeDeviant compares the executions, it will notify the programmer whether the behavior matches between the selected execution and the most recent execution (Fig. 2-E). Once the programmer gets feedback from the tool, he or she can then manually inspect the code if necessary to find the cause of the behavior change.

To better fit into programmers' workflows, CodeDeviant allows for comparisons without any explicit interactions to do so. If the programmer does not choose an execution in the history pane, it will default to the oldest execution in the current development session. This execution was chosen as the default to ensure that it came before the code change. Additionally, CodeDeviant allows for repeated comparisons without any additional actions. That is, the programmer can execute the application over and over, using either the same selected execution to compare with, or by selecting another execution in the history.

C. *Allow Only Apt Comparisons*

As the programmer is performing changes and executing the application, CodeDeviant allows for only apt comparisons since it is possible that there may not be a reasonable way to compare the current execution with the selected previous execution. For example, if the programmer modifies the inputs (e.g., adds an additional parameter), it is not clear how CodeDeviant should compare the executions, and CodeDeviant will provide an error message. Similarly, if the executions do not share any input values, CodeDeviant cannot determine if the behavior is the same and will notify the user.

V. Evaluation Method

To investigate how effectively CodeDeviant helps programmers in detecting behavior-altering refactorings, we ran a within-subjects lab study of programmers refactoring and validating their refactorings. Each participant received two treatments, the control treatment, the standard LabVIEW environment, and the CodeDeviant treatment, an extended version of LabVIEW with CodeDeviant features enabled.

The research questions that we addressed with our empirical evaluation of CodeDeviant were as follows:

- RQ1: Do programmers using CodeDeviant-extended LabVIEW find behavior-altering bugs more *accurately* than programmers using standard LabVIEW?
- RQ2: Do programmers using CodeDeviant-extended LabVIEW find behavior-altering bugs more *quickly* than programmers using standard LabVIEW?
- RQ3: Do programmers consider CodeDeviant to be *helpful*?

Our participants consisted of 12 professional LabVIEW programmers (11 male, 1 female) from National Instruments. They reported, on average, 4.58 years of programming experience ($SD = 1.73$) and 2.23 years of LabVIEW experience ($SD = 1.93$). All participants reported programming in LabVIEW as part of their daily work.

As their primary tasks, the participants performed two refactorings on an existing calculator application written in LabVIEW. They were also asked to verify whether or not the refactoring changed the behavior of the application (but not to perform additional fixes). Each task was based on a refactoring from Fowler's catalog of refactorings [12]. The first task involved replacing two blocks of code with a built-in function (similar to Fowler's Replace Algorithm refactoring). The built-in function behaved differently than the blocks, and thus, in validating the change, the correct answer was that it does change the behavior. The second task involved performing an Extract Method refactoring which should not have changed the program's behavior.

Each session lasted no more than 30 minutes. First, all participants filled out a background questionnaire, and received an introduction to the latest version of LabVIEW and the calculator application. Next, the participants performed the two refactoring tasks where they were free to modify and test the code however they saw fit. Each participant received one treatment for the first task and the other for the second task. Half of the participants were randomly selected to use CodeDeviant-extended LabVIEW first, and the other half used standard LabVIEW first. We asked each participant to "think aloud" as he/she worked. At the end of the session, participants answered a questionnaire regarding the tool and took part in a semi-structured interview. As data, we recorded audio and screen-capture video of each session.

VI. Evaluation Results

A. RQ1 Results: Accuracy of Detecting Bugs

As Fig. 3 shows, when participants used CodeDeviant, they correctly assessed whether their refactorings resulted in changes to the program's behavior far more often than when they used only standard LabVIEW. In fact, when using CodeDeviant, everyone provided the correct answer for the task. In contrast, when using standard LabVIEW, only a third

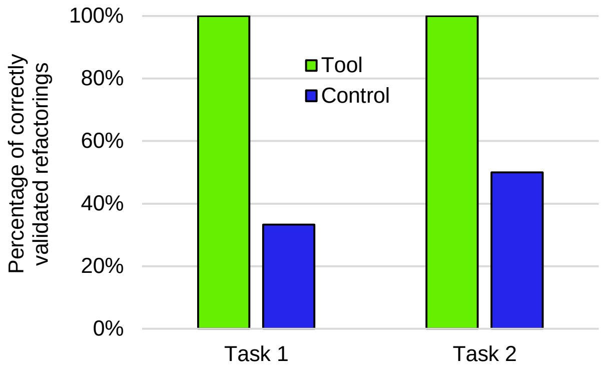


Fig. 3. CodeDeviant users were significantly more accurate in validating their refactorings.

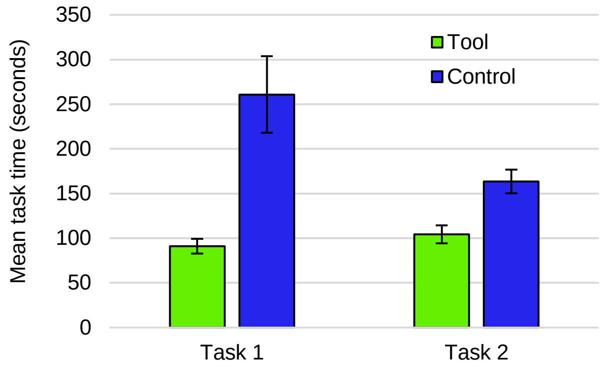


Fig. 4. CodeDeviant users were significantly faster in performing and validating the refactorings (smaller bars are better). Whiskers denote standard error.

of the participants provided the correct answer for the first task and only half for the second task. The differences were significant for Task 1 ($\chi^2(1, N = 12) = 6$, $p = 0.01$) and Task 2 ($\chi^2(1, N = 12) = 4$, $p < 0.05$).

B. RQ2 Results: Time on Task

As Fig. 4 shows, When participants used CodeDeviant, they also completed the tasks considerably faster than when they used only standard LabVIEW. For Task 1, participants using CodeDeviant took roughly a third of the time taken by those using standard LabVIEW. For Task 2, participants using CodeDeviant completed Task 2 over 40% faster than those using standard LabVIEW. A Mann–Whitney U test showed significance for both Task 1 ($U = 0$, $Z = 2.9$, $p = 0.003$) and Task 2 ($U = 2$, $Z = 2.5$, $p = 0.01$).

C. RQ3 Results: Opinions of the Participants

As Fig. 5 shows, the participants generally considered CodeDeviant to be helpful and would use it for their everyday work. Only one participant responded that the tool was not helpful. Furthermore, only two participants said they would not use CodeDeviant if they had it available to them. Details of their concerns are described in the Discussion section.

VII. Discussion

Overall, the results of our CodeDeviant evaluation were notably positive. Participants using CodeDeviant identified

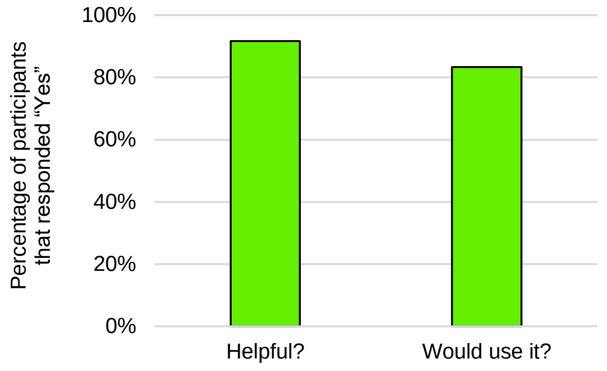


Fig. 5. Participant responses were highly positive on the CodeDeviant opinion questionnaire ("Yes" or "No" answers).

behavior-altering bugs significantly more accurately than those using the control treatment. Moreover, CodeDeviant also helped participants complete the refactoring tasks significantly faster than did the control treatment. In addition to the positive results for these objective performance measures, the participants also expressed predominantly favorable subjective opinions of CodeDeviant.

A. *Qualitative Observations*

To better understand the reasons behind our overwhelmingly positive results, we analyzed our data for qualitative evidence to help explain these outcomes. In particular, we reviewed the participants' comments and mapped from quantitative data points to qualitative episodes of participant behavior, examining those episodes in detail to help explain and expand upon our results.

1) Why Participants Validated Changes More Accurately with CodeDeviant (RQ1): Every participant using CodeDeviant had 100% accuracy in detecting behavior-altering bugs. For every task, participants receiving the CodeDeviant treatment used CodeDeviant to validate their changes, and CodeDeviant reported correctly whether or not the program's output had changed. For example, during P3's second task, he identified the relevant code that needed to be refactored, so he then executed the AppendToDisplay VI twice with two different sets of input. He then performed the code change and reran the VI. Finally, he turned to CodeDeviant, which correctly reported that the behavior did not change.

However, the participants had a much more difficult time when they did not use CodeDeviant. Multiple participants ran the VI one or more times before and after their changes to see if the program's output changed. For example, P5 deliberately executed the program with one set of inputs before making his edits:

P5: "If I care about testing this, then I should go do that first."

Once he finished his change, he reran the application. Although the output was visibly different, he mistakenly declared that it worked the same, perhaps unable to fully recall the original output. Participants P4, P6, and P9 followed similar strategies, and were also unsuccessful in noticing changes in the output of their programs. In one extreme case, P4 alternated between viewing the code and running the application five times prior to making his change, and still failed to notice that his program's output had changed.

Even when participants tried other strategies to validate their changes, they still had difficulties. Participant P10 took a more thorough approach to testing the program by writing down the program output on a piece of paper. However, this note-taking strategy was ultimately unsuccessful. She incorrectly thought that the program's behavior had changed, perhaps because she failed to notice that she had changed the input values between her runs (an inconsistency that CodeDeviant would have caught). In contrast, participant P12 did not rely on running the application at all. Instead, he examined the code, following nearly every wire in RemoveDecimalFromDisplay and ApplyDecimal to understand his change. After examining the wires for over two minutes, he finally declared:

P12: "I'm confident this code does the same thing."

Unfortunately, he was incorrect: the behavior had changed.

2) Why Participants Validated Changes Faster with CodeDeviant (RQ2): Not only were participants more accurate while using CodeDeviant, they also completed the tasks considerably faster. This speedup is likely thanks to the automation provided by CodeDeviant, which eliminated the need to use tedious manual change-validation strategies, such as testing and trying to remember the output values produced before the change, or tracing wires in the hopes of discovering a bug. Using CodeDeviant, participant P10 was able to achieve the fastest overall time for the first task. She began by testing the application, performed the change, ran the application once, and consulted the CodeDeviant output. The whole process took only 85 seconds. Similarly, participant P7 completed the second task in just a little over a minute using CodeDeviant. Thanks to CodeDeviant, he spent the majority of this time on making edits to the code, rather than, say, testing it.

In contrast, when using standard LabVIEW without CodeDeviant, participants took much longer in completing their tasks. The participants' strategies for validating their code seemed to be the main cause of this slowdown. For example, in the case of participant P8's first task, he simply stared at the code for long stretches of time without saying or doing anything. Although his strategy was not entirely clear, he may have been visually tracing the code relevant to his change. He was ultimately correct in reporting that the code change had altered the program's behavior, but it took him over 7 minutes to reach that conclusion.

3) Why Participants Liked (or Disliked) CodeDeviant (RQ3): Participants were generally favorable of CodeDeviant. In the interview, participants expanded on their thoughts of CodeDeviant, providing a range of feedback. Several participants explained how CodeDeviant alleviated the burden of remembering the program behavior as they worked. In particular, participant P11 described his normal working environment, where he can write down his test inputs and outputs on a piece of paper, and how using CodeDeviant will make that process more efficient:

P11: "On the one where I was doing it without having the tool, it isn't terribly difficult to write down... when I'm sitting at my desk I have notepads and pens, but if you can get around having to have one... If you have one output it's fine, but if you have something that has to output an entire array, being able to validate that [with the tool] is really nice."

Other participants had similar sentiments regarding how CodeDeviant can enhance efficiency:

P6: "Now [without the tool] you have to manually see if the output is the same. If you have more inputs or outputs it is harder to do it manually. If you are working on something complex, this would be a really useful tool."

P8: "[Without the tool] it was all on me to remember what I got for the outputs. It isn't too terrible for a small VI but as soon as you hit any level of complexity..."

Furthermore, participants elaborated on the general usefulness of the tool in regards to testing. When asked why they found CodeDeviant useful, P3 and P7 expressed the importance of testing, which they believed CodeDeviant helped with:

P3: "If you don't have to do all the setup yourself, and you just have a tool that will do it for you then I feel like more people would be more willing to do it."

P7: "Having any kind of testing is incredibly important."

Although most participants were favorable of CodeDeviant, one participant said it was not helpful, and two said they would not use it in their daily workflows. Participant P1 reported his concerns about choosing the correct input values:

P1: "What if it only works because these inputs are the ones I'm testing but my change doesn't work."

His concern is valid, but this problem also exists without the tool (e.g., manually providing inputs and observing the outputs). Participant P9 reported that he believed the tool was helpful but would not use it because he believed it might be difficult for others to discover the feature and to integrate into their workflow.

P9: "I think it is a thing that could be useful to people who know about it and are trained to do that, and see the benefit of it... It is kind of a tricky situation to figure out how to help people who don't know how to use the tools that exist."

He later explained that it could be integrated more closely into a programmer's typical workflow:

P9: "It would be neat if it just showed up as a warning in their errors window."

B. Opportunities for Improving CodeDeviant

Based on our participants' feedback and the findings from the user study, we identified several key opportunities for potentially improving the design of CodeDeviant.

1) Interaction Design Improvements: One key opportunity for improvement is to better explain to the programmer *how* executions behaved differently. Currently, CodeDeviant reports whether there is a difference in behavior, but does not communicate how it differed or by how much. For example, CodeDeviant could display the specific input values that resulted in different output values between executions. To provide the programmer more context about what was tested, CodeDeviant could report a correctness percentage (e.g., 75% of the tested values are equal) as well as a coverage percentage (e.g., 20% of the original inputs were tested).

2) Performance Overhead Reduction: A second key opportunity for improvement is by reducing the performance overhead of CodeDeviant. The main overhead stems from recording the program output at runtime. (We did not detect any noticeable performance impact in CPU load while running CodeDeviant.) In our test cases, VIs that ran only once used little storage for recordings (<1KB). However, VIs that ran continuously (certain GUIs and data acquisition functions) used as much as 50MB of storage per minute of recording. While inspecting these data, we observed that over 90% were redundant, and could be filtered periodically to save space.

3) Extended Coverage of Program Behaviors: A third opportunity for improvement is to expand the program behaviors that can be compared by CodeDeviant. Although CodeDeviant never failed to detect a behavior change for of our participants, there are possible scenarios where it could fail. In particular, any application where it is difficult to reproduce the same input values could be problematic. For example, if the application acquires streaming data from specialized hardware (e.g., an instrument for real-time radio measurements) such that each execution will not yield the same input values (or some subset thereof), then CodeDeviant will not be able to do a comparison. To address this problem, CodeDeviant could be enhanced with *replay debugging*, a technique that enables the replaying of events that produced a particular outcome [29].

VIII. Threats to Validity

Our evaluation study has several threats to validity that are inherent to laboratory studies of programmers. The code base was small, and thus, may not be representative of all programs; however, to enhance its realism, we based it on an open source LabVIEW project. Our participants may not have been representative of all expert programmers; however, they were all professional LabVIEW programmers. Reactivity effects, such as the participant trying to please the researchers, may have occurred; however, we attempted to minimize these effects by presenting the two versions of LabVIEW to participants as possible design alternatives, and by not revealing that the CodeDeviant version was the researchers' creation. Finally, our sample size was small, with only 12 participants performing 2 tasks each; however, we used multiple metrics and both quantitative and qualitative analyses to triangulate and enhance confidence in our findings.

IX. Related Work

A. Refactoring Support

Researchers have proposed a variety of automated refactoring tools to improve the efficiency and correctness of these code changes, but they rely on programmers explicitly utilizing these features. However, many studies have shown that the majority of refactorings are performed manually [13], [19], [28], [30], [44]. One notable tool, SafeRefactor [41], generates unit tests for the original code and the refactored code to verify that the behavior does not change when applying automated refactorings. Other work in automated refactoring has been to formally verify the refactoring operations (e.g., [9], [25]) and the refactoring engine (e.g., [26], [40], [42]), and yet there are still bugs introduced by the most commonly used refactoring tools [4], [39], [41], [44], [47]. CodeDeviant was designed to fit into programmers' existing workflow, without requiring them to use automated refactorings.

To better integrate refactoring tools into programmers' existing workflows, researchers have designed tools to assist in manual refactorings, but they rely on detecting when a manual refactoring has occurred. For example, BeneFactor [13], GhostFactor [14], and WitchDoctor [11] aim to recognize when a programmer is performing a refactoring manually and provide features to automatically complete the change while attempting to validate correctness. RefDistiller [1] takes a different approach by identifying manual refactorings after they are completed, and provides features for the programmer to review these changes, while suggesting missing changes and extra changes that are needed to maintain the same behavior. However, these tools rely on static analysis to identify manual refactorings and are limited in the types of refactorings that they support, unlike CodeDeviant, which does not need to detect that the programmer is refactoring.

B. Change Impact Analysis

Change impact analysis is a complementary approach to CodeDeviant's, which locates portions of the code that may be affected by a code change [22]. These analyses could be leveraged by tools to identify whether a refactoring is behavior altering. However, a variety of issues have prevented such techniques from being adopted by programmers in practice. For example, these tools output a list of code locations that are potentially impacted by a change (e.g., EAT [2], Sieve [38], Jimpa [6], JDIA [18], Impala [16], JRipples [5], and ROSE [49]), which then requires a programmer to manually investigate these locations. Another issue is that they can have high rates of false positives and false negatives [16], while more accurate algorithms incur substantial performance costs (e.g., PathImpact [32]). Yet another barrier of these tools is that they may require substantial effort to setup and maintain, such as creating unit test suites (e.g., Crisp [8]) or instrumenting the source code prior to use (e.g., EAT [2]).

C. Program Steering

A related idea to comparing program outputs to validate a code change, is *program steering* [3], which provides continuous feedback to the programmer and allows the programmer to modify the program at runtime. For example, Forms/3 provides affordances to *time travel*, so the programmer can investigate the causes and effects of a program's behavior [3], which could potentially help a programmer detect a bug in their refactoring. Since these features were implemented in a spreadsheet-like environment, it is an open question how well they would generalize to a visual dataflow programming environment (e.g., with streaming data). Additionally, program steering features require considerable changes to a programmer's workflow, which may be a barrier to adoption.

X. Conclusion

In this paper, we have presented the novel CodeDeviant tool design to support programmers in detecting behavior-altering bugs while refactoring visual dataflow code. An evaluation study comparing the CodeDeviant-extended LabVIEW with the standard LabVIEW IDE made the following key findings:

- RQ1 (accuracy): Programmers using CodeDeviant-extended LabVIEW identified behavior-altering bugs significantly more accurately than with standard LabVIEW.
- RQ2 (time): Programmers using CodeDeviant-extended LabVIEW completed the refactoring tasks significantly faster than with standard LabVIEW.
- RQ3 (user opinions): Programmers generally found CodeDeviant helpful and agreed that they would use it in their daily work.

We hope that CodeDeviant and our findings represent a noteworthy advancement toward helping programmers refactor their code more correctly and efficiently. Moving beyond our current work, a promising direction for the future is to explore novel ways in which a programmer can effectively compare all aspects of their program to some previous state of the program, including input values, output values, and code changes. Although there have been tools proposed that provide specific comparisons (e.g., *how did my code look previously?* [17]), there has not been a comprehensive system that allows the programmer to compare all aspects of their program and its behavior. Our CodeDeviant design and implementation demonstrated the strong potential of such a system, eliciting extensive positive feedback and optimism from professional programmers. We believe this work represents a substantial step toward better supporting programmers in the fundamental, yet tedious and error-prone, task of refactoring code.

Acknowledgments

We give special thanks to Andrew Dove for his counsel on all things LabVIEW. This material is based upon work supported by National Instruments and by the National Science Foundation under Grant No. 1302117. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of National Instruments or of the National Science Foundation.

References

[1] E. L. G. Alves, M. Song, and M. Kim, "RefDistiller: A refactoring aware code review tool for inspecting manual refactoring edits," in *Proc. 22nd ACM SIGSOFT Int'l Symp. Foundations of Software Engineering (FSE '14)*, 2014, pp. 751–754.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold, "Efficient and precise dynamic impact analysis using execute-after sequences," in *Proc. 27th Int'l Conf. Software Engineering (ICSE '05)*, 2005, pp. 432–441.

[3] J. W. Atwood, M. M. Burnett, R. A. Walpole, E. M. Wilcox, and S. Yang, "Steering programs via time travel," in *Proc. 1996 IEEE Symp. Visual Languages*, Sep 1996, pp. 4–11.

[4] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs?: An empirical study," in *Proc. 2012 IEEE 12th Int'l Working Conf. Source Code Analysis and Manipulation (SCAM '12)*, 2012, pp. 104–113.

[5] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich, "JRipples: A tool for program comprehension during incremental change," in *13th Int'l Workshop on Program Comprehension (IWPC'05)*, May 2005, pp. 149–152.

[6] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *11th IEEE Int'l Software Metrics Symposium (METRICS '05)*, Sept 2005, pp. 21–29.

[7] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, "Let's go to the whiteboard: How and why software developers use drawings," in *Proc. SIGCHI Conf. Human Factors in Computing Systems (CHI '07)*, 2007, pp. 557–566.

[8] O. C. Chesley, X. Ren, and B. G. Ryder, "Crisp: A debugging tool for Java programs," in *21st IEEE Int'l Conf. Software Maintenance (ICSM'05)*, Sept 2005, pp. 401–410.

[9] M. Cornelio, A. Cavalcanti, and A. Sampaio, "Sound refactorings," *Science of Computer Programming*, vol. 75, no. 3, pp. 106–133, 2010.

[10] B. Daniel, D. Dig, K. Garcia, and D. Marinov, "Automated testing of refactoring engines," in *Proc. the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symp. The Foundations of Software Engineering (ESEC-FSE '07)*, 2007, pp. 185–194.

[11] S. R. Foster, W. G. Griswold, and S. Lerner, "WitchDoctor: IDE support for real-time auto-completion of refactorings," in *Proc. 2012 Int'l Conf. Software Engineering*, 2012, pp. 222–232.

[12] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[13] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proc. 34th Int'l Conf. Software Engineering (ICSE '12)*, 2012, pp. 211–221.

[14] X. Ge and E. Murphy-Hill, "Manual refactoring changes with automated refactoring validation," in *Proc. 36th Int'l Conf. Software Engineering (ICSE '14)*, 2014, pp. 1095–1105.

[15] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.

[16] L. Hattori, G. dos Santos Jr, F. Cardoso, and M. Sampaio, "Mining software repositories for software change impact analysis: A case study," in *Proc. 23rd Brazilian Symp. Databases SBBD '08*, 2008, pp. 210–223.

[17] A. Z. Henley and S. D. Fleming, "Yestercode: Improving code-change support in visual dataflow programming environments," in *2016 IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC '16)*, Sept 2016, pp. 106–114.

[18] L. Huang and Y. T. Song, "A dynamic impact analysis approach for object-oriented programs," in *2008 Advanced Software Engineering and Its Applications*, Dec 2008, pp. 217–220.

[19] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proc. ACM SIGSOFT 20th Int'l Symp. the Foundations of Software Engineering (FSE '12)*, 2012, pp. 50:1–50:11.

[20] A. Kittur, A. M. Peters, A. Diriye, T. Telang, and M. R. Bove, "Costs and benefits of structured information foraging," in *Proc. SIGCHI Conf. Human Factors in Computing Systems*, ser. CHI '13, 2013, pp. 2989–2998.

[21] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, "A case study in refactoring a legacy component for reuse in a product line," in *Software Maintenance, 2005. ICSM'05. Proc. 21st IEEE Int'l Conf.*, Sept 2005, pp. 369–378.

[22] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013.

[23] S. McDonald, "Studying actions in context: a qualitative shadowing method for organizational research," *Qualitative Research*, vol. 5, no. 4, pp. 455–473, 2005.

[24] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb 2004.

[25] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens, "Formalizing refactorings with graph transformations," *J. Softw. Maint. Evol.*, vol. 17, no. 4, pp. 247–276, 2005.

[26] M. Mongiovi, "Safira: A tool for evaluating behavior preservation," in *Proc. ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*, 2011, pp. 213–214.

[27] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?" in *Int'l Conf. Software Reuse*. Springer, 2006, pp. 287–297.

[28] E. Murphy-Hill, C. Parnin, and A. Black, "How we refactor, and how we know it," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 5–18, Jan 2012.

[29] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously recording program execution for deterministic replay debugging," in *Proc. 32Nd Annual Int'l Symp. Computer Architecture (ISCA '05)*, 2005, pp. 284–295.

[30] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A comparative study of manual and automated refactorings," in *Proc. 27th European Conf. Object-Oriented Programming (ECOOP '13)*, 2013, pp. 552–576.

[31] W. F. Opdyke, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Proc. of 1990 Symp. Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, 1990.

[32] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold, "An empirical comparison of dynamic impact analysis algorithms," in *Proc. 26th Int'l Conf. Software Engineering (ICSE '04)*, 2004, pp. 491–500.

[33] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive Psychol.*, vol. 19, no. 3, pp. 295–341, 1987.

[34] D. Piorkowski, S. Fleming, C. Scaffidi, C. Bogart, M. Burnett, B. John, R. Bellamy, and C. Swart, "Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers," in *Proc. ACM SIGCHI Conf. Human Factors in Computing Systems*, ser. CHI '12, 2012, pp. 1471–1480.

[35] D. J. Piorkowski, S. D. Fleming, I. Kwan, M. M. Burnett, C. Scaffidi, R. K. Bellamy, and J. Jordahl, "The whats and hows of programmers' foraging diets," in *Proc. SIGCHI Conf. Human Factors in Computing Systems*, ser. CHI '13, 2013, pp. 3063–3072.

[36] J. U. Pipka, "Refactoring in a test first world," in *Proc. Third Int'l Conf. eXtreme Programming and Flexible Processes in Software Eng*, 2002.

[37] N. Rachatasumrit and M. Kim, "An empirical investigation into the impact of refactoring on regression testing," in *2012 28th IEEE Int'l Conf. Software Maintenance (ICSM)*, Sept 2012, pp. 357–366.

[38] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Sieve: A tool for automatically detecting variations across program versions," in *21st IEEE/ACM Int'l Conf. Automated Software Engineering (ASE'06)*, Sept 2006, pp. 241–252.

[39] M. Schäfer and O. de Moor, "Specifying and implementing refactorings," in *Proc. ACM Int'l Conf. Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*, 2010, pp. 286–301.

[40] M. Schäfer, T. Ekman, and O. de Moor, "Sound and extensible renaming for Java," in *Proc. 23rd ACM SIGPLAN Conf. Object-oriented Programming Systems Languages and Applications (OOPSLA '08)*, 2008, pp. 277–294.

[41] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE Software*, vol. 27, no. 4, pp. 52–57, July 2010.

[42] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 147–162, Feb. 2013.

[43] K. T. Stolee and S. Elbaum, "Refactoring pipe-like mashups for end-user programmers," in *Proc. 33rd Int'l Conf. Software Engineering (ICSE '11)*, 2011, pp. 81–90.

[44] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *Proc. 34th Int'l Conf. Software Engineering (ICSE '12)*, 2012, pp. 233–243.

[45] F. Vonken and A. Zaidman, "Refactoring with unit testing: A match made in heaven?" in *Proc. 2012 19th Working Conf. Reverse Engineering (WCRE '12)*, 2012, pp. 29–38.

[46] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *21st IEEE/ACM Int'l Conf. Automated Software Engineering (ASE '06)*, Sept 2006, pp. 231–240.

[47] P. Weissgerber and S. Diehl, "Are refactorings less error-prone than other changes?" in *Proc. 2006 Int'l Workshop on Mining Software Repositories (MSR '06)*, 2006, pp. 112–118.

[48] K. N. Whitley, L. R. Novick, and D. Fisher, "Evidence in favor of visual representation for the dataflow paradigm: An experiment testing LabVIEW's comprehensibility," *Int'l Journal of Human–Computer Studies*, vol. 64, no. 4, pp. 281–303, 2006.

[49] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, June 2005.