# Yestercode: Improving Code-Change Support in Visual Dataflow Programming Environments

Austin Z. Henley, Scott D. Fleming
Department of Computer Science
University of Memphis
Memphis, Tennessee 38152-3240
Email: {azhenley, Scott.Fleming}@memphis.edu

***Abstract*—In this paper, we present the Yestercode tool for supporting code changes in visual dataflow programming environments. In a formative investigation of LabVIEW programmers, we found that making code changes posed a significant challenge. To address this issue, we designed Yestercode to enable the efficient recording, retrieval, and juxtaposition of visual dataflow code while making code changes. To evaluate Yestercode, we implemented our design as a prototype extension to the LabVIEW programming environment, and ran a user study involving 14 professional LabVIEW programmers that compared Yestercode-extended LabVIEW to the standard LabVIEW IDE. Our results showed that Yestercode users introduced fewer bugs during tasks, completed tasks in about the same time, and experienced lower cognitive loads on tasks. Moreover, participants generally reported that Yestercode was easy to use and that it helped in making change tasks easier.**

## I. Introduction

Visual dataflow programming languages are receiving renewed attention for industrial applications. At their core, these languages represent code using graphical box-and-wire diagrams, where boxes denote functions and wires denote the passing of values between functions. Although textual languages, such as Java and Python, have tended to dominate mainstream programming, a few visual dataflow languages have been able to carve out successful niches—for example, LabVIEW[1] has been used in the domain of science and engineering applications for over 30 years. The benefits of visual dataflow languages have been well documented: easier to learn [3], [5], improved liveness or immediate feedback [25], and more informative notations [11], [28]. A recent surge of visual dataflow languages have aimed to leverage these benefits for a variety of purposes: Google/RelativeWave's Form tool for prototyping phone apps, Filter Forge for producing image filters, RapidMiner for data analysis, and Microsoft Robotics Developer Studio for programming robots.

However, despite this success, visual programming environments have received little or no tool support for making code changes. The tediousness and error-proneness of making code changes, such as refactorings [8], has been well recognized in textual programming environments [13], [17], [18]. Textual programming environments support code changes in various ways, including autocomplete (e.g., Calcite [16] and Graphite [21]), automated refactoring (e.g., ReBA [7] and BeneFactor [9]), and code smell detection (e.g., PMD and DETEX [15]). Although a few researchers have begun investigating code-change support for visual programming environments (e.g., SDPA [6]), the work is still in the early stages, and has yet to see widespread adoption in practice. In our own formative interviews of LabVIEW programmers (Section III-A), many confirmed that making code changes was currently difficult and problematic.

In a formative user study of LabVIEW programmers engaged in refactoring (Section III-B), we found that the programmers encountered considerable challenges during *rewiring*. In visual dataflow languages, rewiring may involve changing existing wires to connect to different box input/outputs, or introducing new wires into existing code. The ability to rewire effectively is of critical importance in visual dataflow languages, because nearly every code change requires some manipulation of wires. However, during rewiring activities, participants consistently forgot what wires represented and introduced bugs by mixing up wires.

To address these problems with rewiring, we propose *Yestercode*, a novel tool concept for visual dataflow programming environments (Section IV). Two key goals of Yestercode are to enable programmers to efficiently access past versions of their code, and to juxtapose a reference copy of an older version with their current code by viewing them side by side. Toward achieving these goals, we made several key design decisions, including transparent automated version logging, tight code-editor integration (e.g., copy and pasting capability from reference-copy to code editor), and annotations to aid comprehension of the differences between the reference copy and current code.

In this paper, we present an initial prototype of Yestercode for LabVIEW and report a user-study evaluation using the prototype (Section V). In particular, our user study involved 14 professional LabVIEW programmers, and addressed the following research questions:

RQ1: Do programmers using Yestercode introduce fewer bugs during change tasks?

RQ2: Does using Yestercode affect the time it takes programmers to complete change tasks (for better or for worse)?

[1]http://www.ni.com/labview/

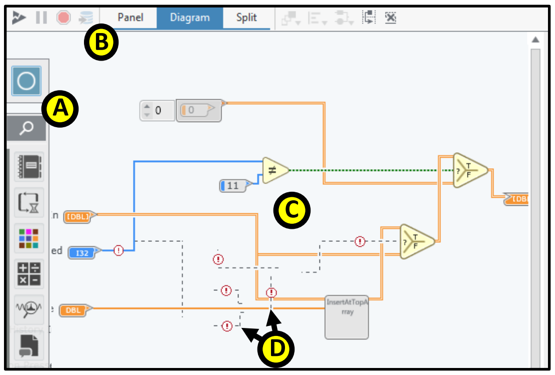

Fig. 1. LabVIEW block-diagram editor. Editors have a palette (A), along with debugging and other controls (B). The pictured editor has open a block diagram (C) that is under construction, with several broken wires (e.g., D).

RQ3: Do programmers using Yestercode have lower cognitive load during change tasks?

RQ4: Do programmers have favorable opinions of Yestercode?

## II. Background: Visual Dataflow Languages

In this work, our aim is to address the problem of code changes in visual dataflow languages. Instead of textual source code, visual dataflow languages are characterized by programs being made up of boxes (functions) and wires (values), as illustrated in Fig. 1-C. The execution of the program follows the dataflow via the wires. Although visual dataflow languages have a radically different syntax than textual languages, such as Java, they still provide many of the same foundational features, such as modularity.

In this paper, we focus on LabVIEW, a commercial visual dataflow programming environment that is one of the most widely used visual programming languages to date [29]. In LabVIEW, programs are composed of modules, called *Virtual Instruments* (VIs). Fig. 1-C illustrates the code for a VI, represented as a *block diagram*. This VI has four inputs (box icons along the left side of the diagram) and one output (box icon at far right). It performs some conditional logic on its input (yellow triangle icons), and calls another VI (gray box; similar to subroutine). This VI is under construction, and contains several broken wires that have one or more disconnected ends and are denoted as dashed lines with red exclamation icons (Fig. 1-D).

## III. Formative Investigation

### A. User Interviews: Modifying Code Is a Problem

As a first step in understanding the barriers that visual dataflow programmers face, we conducted a series of formative interviews. In particular, we held 54 semi-structured interviews with engineers and managers at National Instruments (the maker of LabVIEW). Each interview lasted between 15 and 30 minutes. Our participants consisted of 36 Application Engineers (who provide support to LabVIEW customers), 7 Software Engineers, 4 Hardware Engineers, 4 Group Managers, and 3 UX Engineers. We guided the discussion with questions regarding problems they have with the LabVIEW IDE, problems customers run into, and potential tool support to address these problems.

A key trend from the interviews was that participants found modifying existing LabVIEW code a challenge. Participants often said that either they or customers will go to great lengths to avoid modifying existing code. For example, they often "start from a blank slate" and rewrite code rather than directly modifying code they had previously written. One manager even said that, for her team, modifying code was a "million dollar problem," because several of her engineers waste effort each day rewriting similar but different programs, and then ultimately "throw away the code."

When we prompted participants about *why* modifying existing code was a challenge, their responses provided only a few insights. Most often, they offered vague answers, like that it was "just easier" to rewrite code. A few participants said that it was difficult to remember what specific code does, and that it was too time consuming and tedious to make substantial changes. Three engineers cited the general lack of explicit textual identifiers in visual dataflow code as making it particularly difficult to recall what wires "mean".

### B. User Study: Rewiring Barriers and Coping Strategies

To better understand why programmers said that it is easier to rewrite code rather than modify it in LabVIEW, we conducted an exploratory user study of LabVIEW programmers engaged in change tasks. The study involved 6 participants: 3 graduate students with 1–2.5 years of LabVIEW experience, 1 Hardware Engineer with 4 years of LabVIEW experience, and 2 Software Engineers with 16 years of LabVIEW experience each. Each participant took part in a 60-minute programming session in which he/she made improvements to an existing LabVIEW application that we provided. To get each participant started, we gave him/her three change tasks. Once the participant completed those tasks, we asked him/her to make additional improvements as he/she saw fit. If the participant became stuck, we suggested additional change tasks. At the end of the session, the participant took part in a 30-minute semi-structured interview in which we played back a video of his/her task performance, and asked questions about his/her goals, barriers, and strategies.

A key trend in our user study was that participants exhibited difficulty rewiring components correctly. Many participants made comments about how tedious rewiring a portion of code was. For example, one participant began rewiring, and quickly exclaimed "This is going to take all day!" Rewiring was so difficult that all 6 participants accidentally introduced at least one bug into the program while rewiring. Moreover, 4 of the 6 participants started but then abandoned a code change that they determined was too difficult.

To cope with the challenges of rewiring code, participants commonly used one of three code-change strategies. One strategy was to use the editor's Undo/Redo commands in quick succession to quickly look at older versions of the code being rewired and then return to the current version. Another strategy was, prior to rewiring, to copy and paste the code into the same editor window so as to keep the copy visible as a reference while they rewired the original code. The final strategy was to take screenshots of the code prior to modifying it, and then to use the screenshot as a reference while rewiring. Interestingly, all three of these strategies shared the common theme of enabling the programmers to refer to older versions of the code as they performed rewirings.

## IV. Yestercode Tool Design

Based on the code-change barriers and coping strategies revealed by our formative investigation, we designed the Yestercode tool for visual dataflow programming environments. In particular, Yestercode aims to help with rewiring changes by enabling the programmer to efficiently refer to a prior version of the code while making changes. We hypothesize that by having a readily available reference version of the code, the programmer will exert less mental effort recalling the meaning of the wires and other elements, and thus, will make fewer mistakes resulting from memory failures. Fig. 2 illustrates our Yestercode design (instantiated as an extension to the LabVIEW IDE). We arrived at this design via an iterative process of collecting and incorporating feedback from professional LabVIEW programmers for our evolving design.

As a result of our design process, we identified five key principles for the design of Yestercode. In particular, our design should…

- Transparently record the version history of a block diagram as the programmer edits it.
- Enable efficient navigation of the version history to recover reference versions of the code.
- Enable juxtaposition of the current block diagram with an older reference version of that diagram.
- Provide visual cues in the older reference version of the block diagram to call out differences with the current version.
- Provide tight integration between the reference-version view and the code editor, for example, such that code in the reference version may be copied and pasted into the editor.

In the remainder of this section, we highlight the features of our Yestercode design that satisfy these principles.

### A. Transparent Recording of Version History

As a programmer edits a VI in the code editor (Fig. 2-B, Yestercode automatically records the changes made. To keep the number changes recorded manageable, Yestercode does not record those that concern only the code element's spatial location (e.g., dragging a node to a different location onscreen). This automated recording is transparent to the programmer in that it is completely automatic, requiring no explicit instructions from the user.

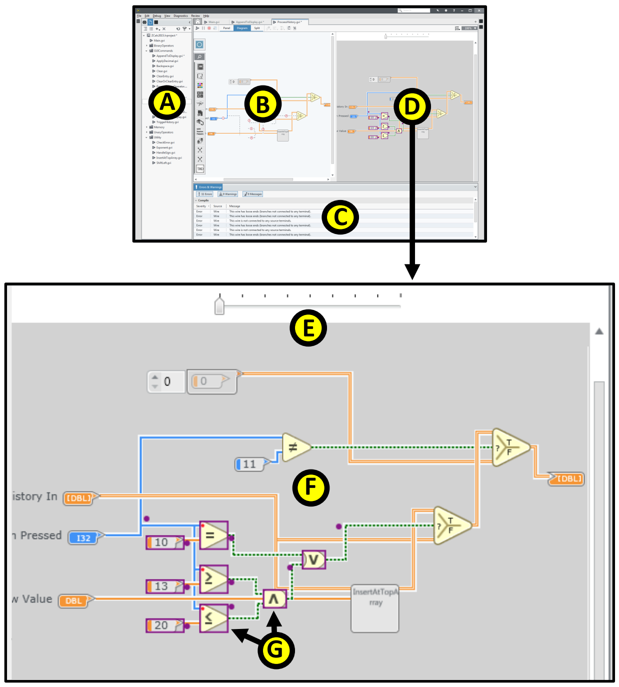

Fig. 2. Yestercode-extended LabVIEW IDE. The standard LabVIEW IDE features include a project explorer (A), a VI code editor (B; see also Fig. 1), and an error-message view (C). Yestercode extends the IDE with an additional view (D) that enables the user to navigate the current VI's version history (E), and that displays the block diagram of the selected older version (F) with annotations denoting differences with the current version (e.g., G).

Based on our formative user study, the design decision to record versions transparently was preferable to having the programmer explicitly initiate the saving of versions. Participants in the user study often did not realize they wanted an older version of the code until they were well into making a change and became stuck. Thus, forcing programmers to explicitly save older versions would introduce a problem of *premature commitment* in which they must make a decision (whether or not to save a version) before they are able to decide (whether they will want to refer to that version) [10].

### B. Efficient Navigation of Version History

To enable efficient navigation of the recorded versions, Yestercode provides a history slider (Fig. 2-E). Sliding the shuttle to the left, steps back in time through the version history. To assist the programmer in finding the appropriate version, Yestercode displays the currently selected version just below the slider (Fig. 2-F). To choose an older version of the code to use as the reference, the programmer needs simply to slide the shuttle to the appropriate version, and leave it there.

### C. Juxtaposition of Current and Older Versions

To enable the programmer to place an older reference version of the code side by side with the current version, Yestercode splits the editor pane to situate a reference-version view (Fig. 2-D) directly next to the code editor (Fig. 2-B). This juxtaposition of editor and reference version enables the programmer to quickly refer back and forth between them, a behavior common to participants in our formative user study. For times when the programmer does not need the reference version, the view can be collapsed and hidden.

### *D. Visual Cues Denoting Version Differences*

To facilitate comparing the features of the reference version of the code with the version in the editor, Yestercode provides visual cues in the reference-version view to indicate boxes and wires that have been removed or updated in the current version (e.g., Fig. 2-G). In particular, a purple border around a box or a purple dot on a wire indicates that the box/wire has been deleted or modified with respect to the current version in the editor. Although Yestercode provides no explicit cues to call out elements that have been added in the current version, based on our experience, it is often the case that existing elements are changed to incorporate new elements, providing cues that implicitly call out the additions.

### *E. Tight Integration with Editor*

To provide tight integration with the code editor, the Yestercode reference-version view is designed to be an editor extension (in our LabVIEW case, sharing the same graphical pane as the editor). In our prototype, tight editor integration offered several benefits. Aside from being read-only, the appearance of and user interactions with the reference-version view were identical to those of the code editor. Chief among these interactions was the ability to copy and paste code from the reference-version view to the code editor. Additionally, other editor features were available in the reference-version view, such as contextual help provided by hovering the mouse pointer over elements in the block diagram, and selection and highlighting of elements in the block diagram to aid reading.

## V. Evaluation Method

To address our research questions (Section I), we conducted a laboratory study of LabVIEW programmers engaged in code-change tasks. The study had two treatments, each for a different version of the LabVIEW IDE. The *control* treatment was associated with use of a standard LabVIEW IDE, and the *Yestercode* treatment was associated with use of a Yestercode-extended LabVIEW IDE. The study had a within-subjects design in which each participant experienced both treatments. To account for order effects, we blocked and balanced based on the treatment orderings.

### *A. Participants*

Our participants consisted of 14 professional LabVIEW programmers (11 male, 3 female) from a large technology company. They reported, on average, 4.36 years of programming experience ($SD = 1.74$) and 1.98 years of LabVIEW experience ($SD = 1.89$). All participants programmed in LabVIEW as part of their daily work.

### *B. Code Base and Change Tasks*

For our study, the participants performed code-change tasks on a calculator application written in LabVIEW and composed of 34 VIs. The application was based on a publicly available sample application[2] written in an old version of LabVIEW. We ported this sample application to the current version of LabVIEW, and modified it to follow the official LabVIEW development guidelines [20]. The code was fully functional and did not contain any known bugs.

Each participant performed six code-change tasks on the calculator application. Each task was based on one of Fowler's refactorings [8]. The tasks were divided into two sequences of three (one sequence for each treatment). For each three-task sequence, the first task required an *Inline Method* refactoring, the second required an *Extract Method* refactoring, and the third required an *Introduce Parameter Object* refactoring. In our LabVIEW context, the Inline Method tasks (IM1 and IM2) each involved replacing a call to a VI with the contents the VI itself; the Extract Method tasks (EM1 and EM2) each involved pulling out part of a VI and making the part its own VI; and the Introduce Parameter Object tasks (IPO1 and IPO2) each involved replacing a set of multiple wires coming out of a block with a single wire that bundles the output values together.

### *C. Procedure*

Each participant took part in an individual session that lasted approximately 1 hour. All participants began the session by filling out a background questionnaire and receiving an introduction to the calculator-application code. Next, each participant completed a first 3-task sequence with one treatment and a second 3-task sequence with the other treatment. Seven randomly selected participants used the control IDE first, and the other seven used the Yestercode-extended IDE first. Each 3-task sequence began with an introduction to the IDE that the participant would be using for that sequence. Next, the participant performed the 3-task sequence in order, completing each task before beginning the next. We asked each participant to "think aloud" as he/she worked. Once the participant had finished each task, he/she completed a cognitive-load questionnaire (details below). At the end of the session, all participants completed an opinion questionnaire regarding Yestercode, and took part in a semi-structured interview in which they discussed any issues that they had while working.

### *D. Data Collection*

The data collected for the study comprised screen-capture video and audio of the participants as well as their questionnaire responses. For the cognitive load questionnaire, we used a well-validated instrument based on Cognitive Load Theory [22]. The instrument features a 7-point Likert question that measures a person's overall cognitive load during a task. Prior work showed that the instrument does not interfere with task performance, is sensitive to small differences in workload, and

[2]http://www.ni.com/example/30779/en/

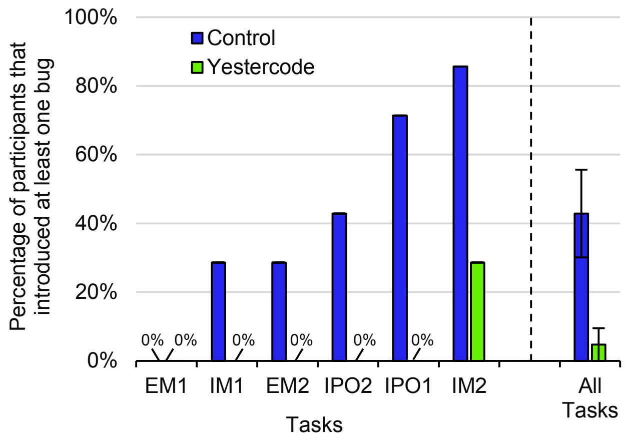

Fig. 3. Yestercode users introduced significantly fewer bugs than control users (smaller bars are better). Whiskers denote standard error.

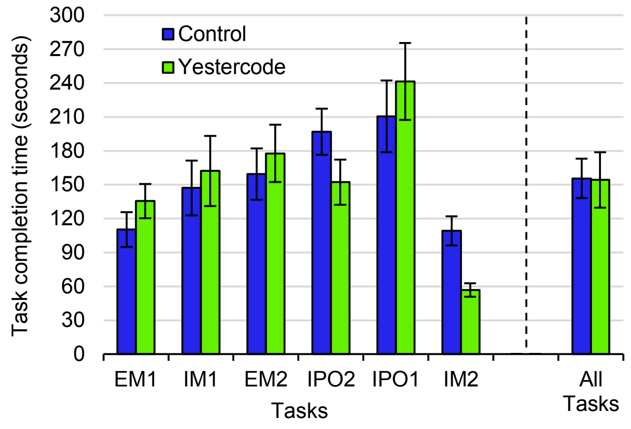

Fig. 4. Overall, control and Yestercode users did not exhibit a significant difference in time taken to complete tasks. Whiskers denote standard error.

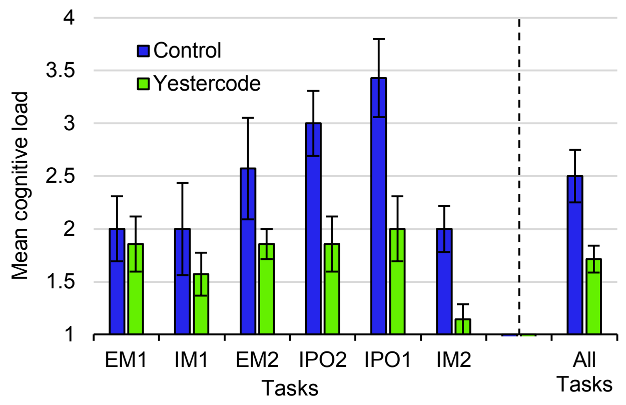

Fig. 5. Yestercode users reported significantly lower cognitive load than control users (smaller bars are better). Whiskers denote standard error.

is reliable [23]. Moreover, the instrument has been shown to highly correlate with more-complex self-reporting instruments (e.g., NASA TLX) [30] as well as with physiological sensors (e.g., heart rate) [23].

## VI. Results

All 14 of our LabVIEW programmers completed all six code-change tasks. Thus, for each task, seven participants completed the task using the control IDE and seven using the Yestercode-extended IDE.

### A. RQ1 Results: Bugs Introduced

As Fig. 3 shows, participants using the Yestercode-extended IDE introduced considerably fewer bugs than those using the control IDE. In fact, Yestercode users did not introduce any bugs, except for one task (IM2). In contrast, control users introduced bugs during every task, but one (EM1). Indeed, the results of a Mann–Whitney $U$ test revealed that Yestercode users introduced significantly fewer bugs than control users ($U = 31.5$, $Z = 2.3$, $p < 0.05$).

### B. RQ2 Results: Time on Task

As Fig. 4 shows, Yestercode users completed tasks in similar amounts of time to that of control users. The task time differed by less than 10% for three of the tasks between the treatments. However, control users took 92% longer than Yestercode users on IM2, which was also the task with the highest rate of bugs. Averaging across all tasks, the time taken to complete the tasks did not differ significantly between the control and Yestercode treatments.

### C. RQ3 Results: Cognitive Load

As Fig. 5 shows, Yestercode users reported considerably lower cognitive loads than control users. In fact, Yestercode users reported a lower cognitive load than control users for every task, on average. For the task that participants introduced the most bugs, IM2, control users reported a cognitive load that was 7 times higher than Yestercode users, on average. The results of a Mann–Whitney $U$ test showed that Yestercode users reported a significantly lower cognitive load than control users ($U = 34.5$, $Z = 2.7$, $p < 0.01$).

### D. RQ4 Results: Opinions of the Participants

To assess participants' opinions of Yestercode, each participant responded to the following questions at the end of the session (7-point Likert scale; "this tool" referred to Yestercode):

- How difficult or easy was this tool to use?
- How helpful was this tool?
- The tasks with this tool were more difficult or easier?

As Fig. 6 shows, participants reported highly positive opinions of Yestercode on the opinion questionnaire. None of the 14 participants responded negatively to any question, and only two participants gave neutral responses (one regarding ease of use and the other regarding whether Yestercode made the tasks easier). Moreover, all participants reported finding Yestercode helpful to some degree.

## VII. Discussion

Overall, the quantitative results of our Yestercode evaluation were highly favorable. Yestercode significantly reduced users perceived cognitive load (Section VI-C). Moreover, Yestercode

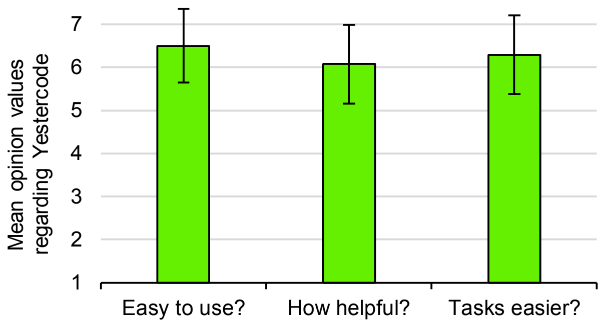

Fig. 6. Participant responses were highly positive on the Yestercode opinion questionnaire (Likert scale from 1 = most negative to 7 = most positive, with 4 = neutral). Whiskers denote standard error.

users took roughly the same amount of time on tasks as control users (Section VI-B), but introduced significantly fewer bugs in their solutions (Section VI-A). Finally, the participants generally scored Yestercode highly on the opinion questionnaire, with no one giving a negative score on any question (Section VI-D).

## *A. Qualitative Observations*

To delve deeper into our results, we analyzed our data for qualitative evidence that helps explain our outcomes.

*1) When participants got stuck, they turned to Yestercode:* Yestercode often helped participants when they got stuck by saving them from having to back out of their changes by undoing. For example, while working on Task IPO1, P4 deleted a group of wires that he was going to rewire. However, having done so, he realized that he no longer knew what they were supposed to be wired to. That is when he turned to Yestercode:

P4: "Oh! I have history! So I deleted all the wires inside the True Case. So instead of Ctrl-Zing, I am going to look at the history."

Similarly, P9 discussed the importance of having Yestercode to help when getting stuck:

P9: "That is the hardest thing when you're refactoring. You delete a bunch of stuff, then you're, like, where did all of it go?"

Other participants echoed P4's sentiment that Yestercode was preferable to using Undo to get back lost information:

P3: "With the history [Yestercode], it was great because I could just click through and I didn't even have to do all those undos."

P6: "When I had the history feature, I never had to undo anything."

*2) Yestercode reduced effort and tedium:* One possible reason that participants expressed preferring Yestercode to using Undo was that Undo was tedious and labor intensive by comparison. For example, when P12 got stuck while using the control IDE, he first looked for a feature like Yestercode. He could not remember where a particular wire went, so he asked the researcher running the session if the control IDE had any capabilities similar to those of Yestercode. After being instructed it did not, he resorted to tediously performing over 15 undo actions to see where the wire originally went. He then clicked Redo another 15 times until he returned to his latest version.

Other participants also complained of having to perform time-consuming and tedious activities in the control IDE to access and manage older versions of the code. For example, P8 expressed his frustration about having to undo his changes:

P8: "No, I don't want to go back to the way it was. I just want to *see* the way it was! I don't want to undo all of my changes just to see it."

Similarly, P10 and P13 described their tedious processes of switching between tabs to see code so that they could perform code changes:

P10: "This was harder to do because I didn't have the history tool [Yestercode]. I had to go back and forth a lot."

P13: "I usually use copy paste when I'm making a change or I'll flip back and forth between two different tabs if I'm moving code."

P2 also expressed preferring the juxtaposition that Yestercode enabled over switching between tabs:

P2: "I think the way it is built, it is over here. It isn't intrusive. You can make it smaller. It is very accessible. It isn't like I have to open up another tab."

To avoid switching between tabs and performing many undo/redo actions, some users of the control IDE took screenshots of the code to use as a reference:

P9: "I use the screenshot tool in Windows to do that for me, so I will take a snapshot, then I will move the window over and edit while I look at it."

P12: "For that first task without history [Yestercode], I had to do a lot of undo and redo just to check. For the next task, I took a screenshot instead."

However, as P7 pointed out, having Yestercode's reference-version view reduced much of the window management that would need to be performed to juxtapose a screenshot:

P7: "With the history [Yestercode], I don't even have to move anything around. I can just see where it goes… It was way easier with this [Yestercode]."

*3) Participants leveraged Yestercode for verification:* One possible reason that Yestercode users produced significantly fewer bugs in their task outcomes was that a number of them used Yestercode to verify their code changes, often at the end of their tasks. One such verification episode was especially fruitful for P10. She believed that she had completed the task but wanted to verify it:

P10: "I could look at this to make sure they're all wired correctly. This is where this comes in handy, for sure."

By performing this verification, she found that several of her wires were incorrect. She then deleted all of the wires, and quickly redid them one by one, checking Yestercode before each change. Similarly, several other participants commented on the utility of Yestercode for verifying correctness:

P12: "This is a good use of the history [Yestercode] where I can go back and make sure something didn't get disconnected. Did all of these wires go to the right place?"

P2: "Even when you don't mess up, it is good for validation."

This tendency to use Yestercode for verification may explain why Yestercode users had similar task times as users of the control IDE. Because Yestercode made it convenient to verify changes, the Yestercode users spent additional time doing so, and also caught more bugs in the process. A similar phenomenon was observed by Burg et al. for their tool, Timelapse [4]: participants used the tool repeatedly, which was viewed as beneficial, yet their task time stayed the same as participants without the tool.

It remains an open question, however, whether Yestercode users had fewer bugs in their task outcomes mainly because of this verification behavior or mainly because they created fewer bugs to begin with. For example, although using Yestercode for verification may have helped some users spot bugs, such use did not guarantee success. There was one instance where a Yestercode participant introduced a bug that went unnoticed, even after using Yestercode to verify his change. In particular, P3 introduced a bug during Task IM2. After performing a code change, he spent nearly a minute reviewing his code using Yestercode. However, he failed to notice that he had accidentally swapped two wires. In sum, although verifying code changes with Yestercode could not catch all bugs, it demonstrably enabled participants to catch some, contributing (at least in part) to the Yestercode users' lower bug-introduction rate.

*4) Participants missed Yestercode when it was gone:* Participants often expressed wishing that they had Yestercode on tasks for which they were assigned the control IDE. For example, while using the control IDE, P2 became confused while performing a rewiring during Task IPO2:

P2: "This is going to be a mess."

As a result, he undid all his work and laboriously repeated it. Later, when he was introduced to the features of Yestercode, he lamented the time he could have saved earlier if he had had the tool:

P2: "In the last one [task], I was deleting wires and couldn't remember what they were. I had to undo everything."

Other participants also made comments on how helpful Yestercode would have been on tasks they were made to perform with the control IDE:

P7: "This [Yestercode] would have been helpful on the first tasks."

P5: "On the first three tasks, I was completely deleting everything and it would have been nice to just look to my right [at Yestercode]."

P8: "This one [later task] was more difficult, because I didn't have the history window [Yestercode]. On the other one [earlier task], I knew I could just go back and rely on that. With this one, I had to plan everything, or else I would get jammed."

P13: "This [Yestercode] would have been helpful on the previous exercises to see where the wires were and where they go."

## B. Related Work

In prior work, programmers have needed access to older versions of code for a variety of reasons. Researchers have studied programmers in the contexts of backtracking to a previous version of code [31], aborting a refactoring [26], and exploring the design space or variations of a program [14], [24]. Our motivation differs from that of these works in that our purpose in storing and retrieving older versions is to provide the programmer with a reference while making changes to the code.

The two most prevalent technologies for storing and retrieving older versions of code are version control systems and undo features. Version control systems (VCSs) are widely used in professional software development for managing versions of source code and facilitating collaborative development. For example, Git is one such VCS that has been growing in popularity, with one website reporting that there are 266,781 public Git repositories[3]. However, most VCSs require users to explicitly perform commit actions when they want to save a revision of their code. As mentioned in Section IV, in our context of code-change support, this user interaction introduces a premature commitment issue [10]: the programmer must know ahead of time which versions he/she will need later, and thus, which to versions to save using their VCS.

Using undo features are also common for backtracking to a previous version of code. This invaluable feature is common in many applications and allows a user to change his/her code file by a single step in a linear fashion [2]. A longitudinal study found that programmers backtracked more than 10 times per hour using undo features or manually changing the code [31]. Even our formative user study participants would often use undo to get back to a previous reference version of their code. However, in doing so, they would temporarily give up their current version of the code in order to view a previous version, and they ran the risk of losing their current version permanently if they accidentally made a change after performing undo. Researchers have proposed a number of approaches to address this problem, including ones based on a tree-based history (e.g., [27]) and selective undo (e.g., GINA [1], Amulet [19], and Azurite [32]); however, none of these approaches were specifically designed to address the issue of providing a reference version to compare with the current version.

Beyond getting back to a previous version of code, programmers often want to compare two versions side-by-side. The UNIX *diff* utility is the seminal tool for comparing two text files. Its features have been incorporated into mainstream code editors, such as Eclipse and Visual Studio, to allow the comparison of two versions of code, often with text highlighting to indicate inserted or deleted text. Furthermore, researchers have applied diff to programming concepts other than text, such as LSdiff, which identifies systematic code changes by analyzing code elements and their structural dependencies [12]. Indeed,

[3]https://www.openhub.net/repositories/compare

Yestercode's annotations for denoting differences between the reference block diagram and current one were inspired by diff.

More recently, researchers have combined undo support with a comparison view. Azurite is an Eclipse plug-in for textual-code editors that allows a programmer to selectively undo a region of code to a previous state, and provides a comparison feature that shows a before and after version of the relevant code [32]. Additionally, Azurite transparently logs the code edits and shows a timeline visualization. While this tool fully supports programmers trying to backtrack, it may not effectively support a programmer who is attempting to perform a new change while referencing a previous version. For example, unlike Yestercode, the comparison view is in an interface separate from the code editor, and as a consequence, the programmer cannot make edits to the current version while juxtaposing it with the older version.

### *C. Limitations*

Our user study has several limitations inherent to laboratory studies of programmers. First, our sample of visual dataflow programmers and the tasks in our study may not represent all such programmers; however, we recruited professionals from a large tech company to increase the likelihood that they are representative of other professional programmers. Second, the code base was small, but we based it on an open source project in an effort to make it more representative of actual projects. Third, the tasks, although based on common refactorings from the literature, were relatively short, and may not be representative of more complex tasks. Fourth, reactivity effects (e.g., participants consciously or unconsciously trying to please the researchers) may have occurred; however, we tried to minimize them by presenting the two versions of LabVIEW as possible design alternatives for a new release, and did not disclose that the Yestercode version was our invention. Finally, order effects of the tool and tasks may have affected our observations, but we counterbalanced the tool order to control for this effect with respect to our statistical tests.

## VIII. Conclusion

In this paper, we introduced the novel Yestercode tool to support rewiring code changes in visual dataflow languages. Our formative investigation revealed that visual dataflow programmers had substantial problems rewiring code during change tasks (Section III). Following an iterative design process, we designed the tool around five key principles (Section IV), and implemented a prototype of it as an extension to the LabVIEW IDE. An evaluation study comparing our Yestercode prototype with a standard LabVIEW IDE made the following key findings:

- RQ1 (bugs): Programmers using Yestercode introduced significantly fewer bugs during change tasks.
- RQ2 (time): Yestercode had no noticeable effect on task time.
- RQ3 (cognitive load): Programmers using Yestercode experienced significantly lower cognitive load during change tasks.
- RQ4 (user opinions): Programmers generally found Yestercode easy to use and helpful in making change tasks easier.

We hope that Yestercode marks a substantial step toward comprehensive support for change tasks in visual dataflow languages. In the future, Yestercode could be extended in a variety of ways to further improve the support that it provides, such as more-intelligently grouping small edits or providing a visualization-enhanced version timeline. More-radical changes could include enabling the branching of code versions and the editing of different versions in parallel. Additionally, our studies have revealed other areas ripe for investigation, for example, how to support programmers in the comparison of runtime behavior between versions of code in order to assist them in understanding how their changes impact the program's behavior. These ideas have considerable potential to dramatically improve the productivity as well as enjoyment of visual dataflow programmers—like Yestercode was able to do for our Participant P5:

P5: "Neat! I don't have to undo, I can just use my history feature."

## Acknowledgments

We give special thanks to Andrew Dove for his counsel on all things LabVIEW. We would also like to thank YoungSeok Yoon for his helpful feedback on a draft of the paper.

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] T. Berlage, "A selective undo mechanism for graphical user interfaces based on command objects," *ACM Trans. Comput.-Hum. Interact.*, vol. 1, no. 3, pp. 269–294, Sep. 1994.

[2] T. Berlage and A. Genau, "A framework for shared applications with a replicated architecture," in *Proceedings of the 6th Annual ACM Symposium on User Interface Software and Technology (UIST '93)*. ACM, 1993, pp. 249–257.

[3] A. Blackwell, "Metacognitive theories of visual programming: What do we think we are doing?" in *Proceedings of the IEEE Symposium on Visual Languages (VL '96)*, 1996, pp. 240–246.

[4] B. Burg, A. J. Ko, and M. D. Ernst, "Explaining visual changes in web interfaces," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. ACM, 2015, pp. 259–268.

[5] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, and P. van Zee, "Scaling up visual programming languages," *Computer*, vol. 28, no. 3, pp. 45–54, Mar. 1995.

[6] C. Chambers and C. Scaffidi, "Smell-driven performance analysis for end-user programmers," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*. IEEE, 2013, pp. 159–166.

[7] D. Dig, S. Negara, V. Mohindra, and R. Johnson, "ReBA: Refactoring-aware binary adaptation of evolving libraries," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, 2008, pp. 441–450.

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

[9] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, 2012, pp. 211–221.

[10] T. R. G. Green, "Cognitive dimensions of notations," in *People and Computers V*. Cambridge University Press, 1989, pp. 443–460.

[11] 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.

[12] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, 2009, pp. 309–319.

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

[14] S. K. Kuttal, A. Sarma, and G. Rothermel, "On the benefits of providing versioning support for end users: An empirical study," *ACM Trans. Comput.-Hum. Interact.*, vol. 21, no. 2, pp. 9:1–9:43, Feb. 2014.

[15] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[16] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers, "Calcite: Completing code completion for constructors using crowds," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '10)*, 2010, pp. 15–22.

[17] E. Murphy-Hill and A. P. Black, "Breaking the barriers to successful refactoring," in *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering (ICSE '08)*. IEEE, 2008, pp. 421–430.

[18] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," in *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, 2009, pp. 287–297.

[19] B. A. Myers, R. G. McDaniel, R. C. Miller, A. S. Ferrency, A. Faulring, B. D. Kyle, A. Mickish, A. Klimovitski, and P. Doane, "The Amulet environment: New models for effective user interface software development," *IEEE Transactions on Software Engineering*, vol. 23, no. 6, pp. 347–365, 1997.

[20] *LabVIEW Development Guidelines*, National Instruments, Apr. 2003, Part Number 321393D-01. [Online]. Available: http://www.ni.com/pdf/manuals/321393d.pdf

[21] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers, "Active code completion," in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, 2012, pp. 859–869.

[22] F. Paas, J. E. Tuovinen, H. Tabbers, and P. W. Van Gerven, "Cognitive load measurement as a means to advance cognitive load theory," *Educational Psychologist*, vol. 38, no. 1, pp. 63–71, 2003.

[23] F. G. Paas, J. J. Van Merriënboer, and J. J. Adam, "Measurement of cognitive load in instructional research," *Perceptual and Motor Skills*, vol. 79, no. 1, pp. 419–430, 1994.

[24] S. Srinivasa Ragavan, S. K. Kuttal, C. Hill, A. Sarma, D. Piorkowski, and M. Burnett, "Foraging among an overabundance of similar variants," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. ACM, 2016, pp. 3509–3521.

[25] S. L. Tanimoto, "VIVA: A visual language for image processing," *Journal of Visual Languages & Computing*, vol. 1, no. 2, pp. 127–139, 1990.

[26] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, disuse, and misuse of automated refactorings," in *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, 2012, pp. 233–243.

[27] J. S. Vitter, "US&R: A new framework for redoing," in *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE '84)*. ACM, 1984, pp. 168–176.

[28] K. N. Whitley, "Visual programming languages and the empirical evidence for and against," *Journal of Visual Languages & Computing*, vol. 8, no. 1, pp. 109–142, 1997.

[29] 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," *International Journal of Human–Computer Studies*, vol. 64, no. 4, pp. 281–303, 2006.

[30] D. Windell and E. Wiebe, "Measuring cognitive load in multimedia instruction: A comparison of two instruments," *Annual meeting of the American Educational Research Association*, 2007.

[31] Y. Yoon and B. A. Myers, "A longitudinal study of programmers' backtracking," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '14)*, 2014, pp. 101–108.

[32] Y. Yoon and B. A. Myers, "Supporting selective undo in a code editor," in *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. IEEE Press, 2015, pp. 223–233.