Utility and accuracy of smell-driven performance analysis for end-user programmers

Citation	Chambers, C., & Scaffidi, C. (2015). Utility and accuracy of smell-driven performance analysis for end-user programmers. Journal of Visual Languages & Computing, 26, 1-14. doi:10.1016/j.jvlc.2014.10.017
DOI	10.1016/j.jvlc.2014.10.017
Publisher	Elsevier
Version	Accepted Manuscript
Terms of Use	http://cdss.library.oregonstate.edu/sa-termsofuse

Utility and accuracy of smell-driven performance analysis for end-user programmers

Christopher Chambers, Christopher Scaffidi*
{ chamberc, scaffidc} @onid.orst.edu

School of Electrical Engineering and Computer Science
Oregon State University
1148 Kelley Engineering Center
Corvallis, OR 97331

*=corresponding author, 541-737-5572

An earlier version of this paper appeared as the following: Chambers, C., and Scaffidi, C. (2013) Smell-driven performance analysis for end-user programmers, *IEEE Symposium on Visual Languages and Human-Centric Computing*.

The current paper expands on the earlier paper by providing (1) an enhanced detection method called smell-driven profiling that also incorporates runtime analysis in addition to the static analysis presented in the earlier work, and (2) a new study assessing how accurately the original and enhanced methods can identify non-trivial performance problems.

Abstract

This paper proposes a technique, called Smell-driven performance analysis (SDPA), which automatically provides situated explanations within a visual dataflow language IDE to help end-user programmers to overcome performance problems without leaving the visual dataflow paradigm. An experiment showed SDPA increased end-user programmers' success rates at finding performance problems and decreased the time required for finding solutions. Another study, based on using SDPA to analyze a corpus of example end-user programs, revealed that it is usually accurate at identifying performance problems. Based on these results, we conclude that SDPA provides a reliable basis for helping end-user programmers to troubleshoot performance problems, as well as a potential foundation for future work aimed at training users and at aiding code reuse.

Keywords: end-user programming; performance; visual language

Highlights

- Smell-Driven Performance Analysis (SDPA) finds dataflow performance problems.
- SDPA provides situated explanations within the visual dataflow language.
- We present an extended form of the technique that incorporates runtime profiling.
- In a user study, participants could more easily diagnose performance problems.
- A second study confirmed that profiling improves accuracy.

1. Introduction

Professional software engineers have long known that the performance of a program can have a powerful effect on its value. Some end-user programmers also share this concern about performance. For example, empirical studies have revealed that scientists tend to rely on textual general-purpose languages such as C++ or Fortran for high-performance computing, despite the availability of domain-specific languages designed for high usability [2][10][22].

LabVIEW is one example of a programming language that has high usability but sometimes leads to programs with inadequate performance. Its maker, National Instruments, claims LabVIEW is the "most widely used development environment for instrument connectivity and [hardware] test application," particularly among engineers and scientists [12]. An independent survey of LabVIEW users investigated the reasons for this high level of adoption, finding that users appreciated LabVIEW primarily due to its visual dataflow language and secondarily due to its support for code reuse [26]. One respondent summarized, "The development time for LabVIEW is less than half that for C," while another claimed to be "3X more productive than programming in C". Yet at the same time, survey respondents sometimes found LabVIEW performance to be inadequate. They expressed concerns that LabVIEW offered no way to optimize usage of registers, memory, disk, and CPU. The researchers conducting the survey noted, "LabVIEW solves these problems by allowing the programmer to call code written in other languages"—in other words, the "solution" to the problem is to step outside LabVIEW.

These challenges are not unique to LabVIEW. Nardi lists HP VEE and Prograph as two other canonical examples of visual dataflow languages [16], where data input/output nodes are connected to one another via computation nodes and virtual dataflow "wires." Scientists and engineers using these languages (e.g., [11][23]) also encountered performance problems. To solve such a problem, "Parts of the application were written in the C language and linked to the VEE application to obtain higher throughput" [11].

In short, these empirical studies reveal a way in which visual dataflow languages can present a "low ceiling" over what users can accomplish [15]. To date, researchers have mainly considered language ceiling from the standpoint of *functionality*. For example, one paper discussed how end-user programmers might encounter a ceiling when they try to create custom widgets [15], while another discussed the ceiling that people encounter when they go from animating 2D images to animating 3D images [20]. In contrast, the empirical studies above reveal a ceiling in terms of *how well* programs carry out functions, rather than in terms of *what* functions are carried out. Helping end-user programmers to overcome performance problems and break through this ceiling—*without* leaving the visual dataflow language—is an unexplored challenge.

In this paper, we propose a new technique called *Smell-driven performance analysis* (SDPA) aimed at meeting this challenge. The technique uses the established concept of a "bad smell," which is a heuristic for finding sections of code that function correctly but that have poor quality [5], and applies this concept in the context of visual dataflow languages favored by some end-user programmer populations. Specifically, *Smell-driven performance analysis* involves statically analyzing programs to heuristically detect areas with potential performance problems, optionally combining static analysis results with a profiling-based dynamic analysis (smell-driven profiling) that identifies areas in the code that are consuming significant execution time ("hotspots"), alerting end-user programmers about problems, and advising on how to fix those problems. The feedback is provided through situated explanations inside the language's IDE, which means that the users will not have to switch windows to find the information and can remain in the visual language or IDE that they are comfortable working in.

To test this approach, we have created a proof-of-concept tool that applies our proposed technique to LabVIEW. To assess utility, we conducted a user study with programmers (skilled LabVIEW users) involving tasks that called for troubleshooting performance problems that we derived directly from real-world programs posted in online forums. The experiment showed that participants were more successful at finding performance problems with our prototype than without it, and having found a problem, they were faster at finding the correct solution. We tested the accuracy of this approach for finding non-trivial performance problems by retrieving a corpus of real programs that users had posted onto the LabVIEW online forums with requests for help at improving performance. We found that most of the issues with the code identified by our prototype did, in fact, indicate real cases where users' code structures caused non-trivial impacts on performance in terms of CPU or memory usage. We found that activating the dynamic analysis enhancement of our technique further improved accuracy.

The remainder of this paper is organized as follows. Section 2 presents a formative study aimed at grounding our approach. Section 3 outlines the techniques of our approach in general terms, and Section 4 describes the specific implementation we created for evaluating the approach with LabVIEW. Section 5 presents the user study that showed high potential utility for users, while Section 6 discusses a second study showing that the approach does an accurate job of finding performance problems on a wide range of real user programs. Sections 7 through 9 summarize threats to validity, related work, and future work, respectively.

2. Preliminary formative study

In our formative study, we interviewed LabVIEW technical support personnel to investigate the problems that LabVIEW users encounter. We were particularly interested in knowing whether many performance problems primarily result from how end-user programmers implement programs.

2.1 Participants and study methodology

The participants for our study were Application Engineer (AE) Specialists. AE Specialists answer customer questions about LabVIEW and assist them in troubleshooting programs. Becoming an AE Specialist requires an ordinary technical support staff member (an AE) to undergo additional training and experience (e.g., in development of custom controls/widgets, or in test automation). AE Specialists are assigned the hardest problems, which typically are too hard for AE staff. Thus, AE Specialists are elite personnel and are qualified to comment on problems encountered by LabVIEW programmers.

National Instruments provided an email list of 13 AE specialists, among whom we successfully recruited 9 for 30-minute interviews (6 via phone and 3 in-person).

The interview had several parts:

- We asked participants how many years of experience they had with LabVIEW, what roles they have had at National Instruments, and how many issues they handle per month.
- We asked them to identify and describe a problem they had recently helped to solve, as well as the needed solution.
- We asked them to consider more generally what problems users most frequently experience in a long list of different areas. These questions covered design problems, program structure problems, problems with specific LabVIEW programming primitives, debugging problems, problems with error-handling, driver and other platform platforms, and problems in the development environment.

To analyze answers about user problems, we used thematic analysis, a grounded content analysis method that identifies recurring topics [13]. We then counted how many times each issue was mentioned. A second researcher reviewed each categorization to verify that it appeared appropriate.

2.2 Results from the preliminary study

All participants had at least 3 years of experience, with the average just under 5 years. Each had served as an AE for 1-2 years before becoming an AE Specialist. Most reported their workload had been 5-10 straightforward support issues per day as an AE, and that their workload as an AE Specialist was approximately 10 relatively complex issues per month.

Participants' answers revealed a long list of recurring problems that, for the most part, resulted directly from the structure of end-user programs. These structures were almost always problematic because of their impact on the performance of programs. For example, the Build Array in Loop problem (Table 1) refers to the situation where a Build Array primitive (which allocates memory) is nested inside of a Loop primitive, which causes frequent memory allocation and de-allocation; this, in turn, causes memory fragmentation, unnecessary memory collection overhead, and ultimately poor execution time for the program as a whole. As another example, the Too Many Variables problem refers to a situation where one virtual instrument (VI) contains more than a few variables. A VI is a single block of computation, analogous to a function. AE Specialists indicated that people sometimes get confused about how to keep variable values consistent when many variables are used, and in code that compiles to multiple parallelized loops, this can result in race conditions.

Overall, 10 of the 13 problematic structures listed in Table 1 had direct implications for performance according to interviewees. These implications were not minor, either: All of these code structures had such

an obvious, often-catastrophic impact on performance that users had taken the time and expense to talk with technical support.

Performance problems manifested at runtime in a variety of ways that included missed deadlines, race conditions, jitter, dropped samples, system crashes, and data corruption. In one example, a LabVIEW user contacted NI about a program that would at first run fine, but after 10-15 minutes of execution would slow down and eventually blue-screen the computer. To try and fix the problem, the user kept increasing the virtual memory size, until this solution was no longer viable. After a brief code review, the program was found to be continually creating duplicates of a 650 MB array (Multiple Array Copies). Each duplicate caused the memory usage of the program to increase until the computer crashed. AE Specialists told many such anecdotes about how the code structures in Table 1 led to catastrophically bad performance.

In addition to performance problems caused by end-user programmers' code, the AE Specialists also told us about 3 problems that had no performance ramifications. These were basic programming issues, such as the "Unconnected Front Panel elements" problem that involves simply forgetting to connect user interface elements of the program (i.e., parts of the "front panel") to computational elements of the program.

One topic that interviewees did not mention was a unified resource for finding and solving performance problems. The names of problems, in Table 1, are names that we assigned; even though different AE Specialists often saw the same problems, they had no standardized names for problems, let alone a consolidated reference for characterizing problems. Instead, they relied on experience and intuition to diagnose trouble. Lacking such expertise, end users' choices are to purchase technical support, buy better hardware, or rely on available training materials.

We reviewed the training materials that LabVIEW trainers provide in introductory seminars, and we identified two other potential smells:

- Uninitialized Shift Registers: LabVIEW has a loop construct that accepts a value (called a "shift register") as input to each iteration, then computes a new value which is used as the input to the next iteration. When the input has no explicit initializer in the code, then a default value is used for the first execution, and the last output from the previous execution is used as the initial value for subsequent executions. Training materials warn that if this value can consume substantial memory of it is an array or other structure that grows during each execution.
- Terminals inside Structures: In LabVIEW, a terminal is an output to the user interface. Many user interface controls are not thread-safe, so terminals implicitly wrap an exclusive lock around each user interface element. Putting a terminal inside of a loop structure causes the program to repeatedly acquire and release the user interface lock. This can reduce performance substantially, causing an otherwise perfectly reasonable program to crawl. Worse, during the execution of this loop, any other areas of the code trying to write to that user interface control will block, and vice versa. As a result, other areas of the program can also substantially slow down due to lock contention.

In addition to these two issues, training materials also mentioned several of the same concerns that our interviewees had discussed. For example, Multiple Array Copies, where a value is copied and passed to multiple nodes in a program, is a well-known problem discussed in several places. Yet, given feedback from the technical support staff, it is clear that many users nonetheless make this mistake in their programs. Clearly, users continue to need help with finding and fixing performance problems in their code.

3. Smell-driven performance analysis

The formative study revealed that many performance problems do result from end-user programmers' code. Below, we describe our technique at the conceptual level, from the standpoint of an end-user programmer who needs help with solving a performance problem. We defer discussion of technical implementation details to the next section.

Finding problems by detecting performance smells: Suppose an end-user programmer has a certain program and is unsatisfied with some aspect of performance. For example, the program might be missing real-time deadlines or dropping samples. We envision that the programmer will open up the problematic program in the development environment for the visual dataflow language. This environment would be enhanced with a new tool that aids in analyzing performance.

Such a tool would take advantage of the fact that the causes of many performance problems are specific structures in the source code (as shown in Table 1). Some of these, such as too many variables or too deeply-

nested loops, resemble canonical "bad code smells" used to characterize object-oriented code that appears to be of poor quality (typically because it is hard to maintain) [5]. In object-oriented code, smells are found using heuristics, such as a rule that any method with more than some N lines of code is potentially poor quality. Likewise, a tool in our technique would statically analyze the program to detect areas matching heuristics that we call "bad performance smells." Each heuristic would describe one common structure that often causes problems (e.g., each performance-related structure listed in Table 1 for LabVIEW).

The next step in our technique is for the tool insert visual alerts (e.g., icons) into each area of code that matches a heuristic. To avoid annoying users with uninvited advice, our tool by default would not actually show alerts unless the user asked for help finding a performance problem (e.g., by clicking a "Diagnose Performance" button). A configuration setting could allow users to continuously run the tool in the background, with any alerts appearing as passive notifications.

Filtering problems with profiling: There is no guarantee that the detected smells are actually causing performance problems. If our tool gives users too many false positives, smells that do not actually cause a performance problem, it is very likely that they would grow frustrated with our tool and stop using it. To help alleviate this potential problem, our tool can provide users with the ability to profile their code and find the actual hot spots.

If the user selects the "profile code" option, the tool will first begin by gathering timing and memory data for every smell that has been detected in the program, as well as the total execution time and total memory usage of the program. Once the program has finished executing, the information for each smell can be compared to the information for the program. Any smell that takes up a large proportion memory or time can be said to be a "True Positive" as it actually causes a performance problem. The smells that are profiled by the tool that do not greatly impact performance would be filtered out by the tool, causing the corresponding visual alerts to disappear from the user interface.

Advising on how to fix problems: Finally, the tool would provide situated advice within the visual language IDE about how to fix problems. Specifically, if a user clicks a visual alert, the tool would summarize the heuristic that had led to that alert. It would explain relevant programming concepts and describe a code modification aimed at eliminating the problematic structure (ideally without altering the code's functionality). Fixing a problem would make the alert disappear (or the user could just dismiss it).

The technique is explicitly designed to be compatible with how users prefer to debug their code. Specifically, an empirical study of end-user programmers found that their favorite debugging strategies included (1) inspecting code, (2) executing with test inputs, and (3) tracing dataflow [24]. Our technique is well-situated to support performance debugging during programmers' performance of these strategies because it (1) presents alerts directly linked to the nodes that users would look at during code inspection, (2) integrates profiling information during execution with test inputs, and (3) only requires a dataflow representation of code in order to find and explain performance problems.

In terms of user interaction, our proposed technique resembles Surprise-Explain-Reward, whereby a tool helps users to find and fix spreadsheets' computational errors [27]. In that approach, users can enter a testing mode, within which the tool shows question marks alongside elements of the program (spreadsheet cells) on the screen to pique curiosity. When the user responds, the tool explains how to test part of the spreadsheet. Users are rewarded for testing by changes in the user interface. Our technique uses alerts to draw attention to targeted parts of a program, presents situated advice to explain how the user should respond, and hides alerts to reward user responses. Our technique differs from Surprise-Explain-Reward in that it is guided by heuristics (smells) and is oriented toward helping users with fixing performance problems rather than fixing computation errors (functionality).

4. Prototype tool for LabVIEW

We have implemented a prototype tool to apply our technique on LabVIEW code. This tool is fully integrated with the version of the LabVIEW environment currently under development at National Instruments. This integration enabled us to evaluate the effectiveness of our technique in a realistic end-user programming environment. Even though we used LabVIEW for this work, we see no reason why we could not create similar tools for other visual dataflow languages, although presumably the specific heuristics needed would vary. Below, we describe the smell-based static analysis implementation, as well as the enhanced version that incorporates runtime profiling information.

4.1 Smell detection via static analysis

Our prototype begins by acquiring a data structure representing the LabVIEW program at hand. This is done by taking advantage of the existing DFIR graph, which is LabVIEW's hierarchical, graph-based internal representation of a program's code [17]. The DFIR graph of a program is very similar to the program's visual representation on the screen, except (1) the DFIR graph decomposes some program elements that appear as single icons in the visual program, and (2) the DFIR graph includes optimizations achieved by minor program transformations (e.g., dead code elimination). In all other ways, however, a DFIR graph is to LabVIEW what an abstract syntax tree is to Java or a similar textual language.

Next, the prototype runs bad performance smell detectors on the program. Each detector checks for one of the performance problems in Table 1 by implementing a heuristic for finding that problem. The detector does this by visiting each node of the DFIR graph relevant to a particular smell, then computing which nodes (if any) meet the heuristic. For example, our detector finds instances of Build Array in Loop smells by visiting all Build Array nodes in the DFIR graph, then for each Build Array node, recursively working outward in the DFIR graph (analogous to working up an abstract syntax tree) to check if the Build Array Node is nested inside a Loop node; in this case, the heuristic is simply checking whether a Build Array node is anywhere inside a Loop node. Likewise, the Too Many Variables smell detector visits all variable-declaration nodes and counts to check if more than 5 are in any one VI.

Our prototype alerts the user to detected smells by annotating program elements with small triangle icons, as shown in Figure 1. (Although Figure 1 shows our tool alert in LabVIEW 2012, we actually ran our tool during our study in a newer version of LabVIEW that is not yet commercially available; National Instruments requested we show the old version here to protect their new user interface's confidentiality.) Since smells usually involve more than one node, there is the question of which nodes should receive these alert annotations. Our rule of thumb is to place an alert node on each program element that an end-user programmer would need to modify to fix the smell; usually, this is one program element or a handful of elements. This method provides situated explanations to the users so that they will not have to change the window or lose focus on their current task.

In addition to the alerts that appear on the visual code itself, our prototype also shows a tabular list of all smells found in the program at the user's request (which would appear, for example, beneath the program in Figure 1). This table succinctly explains each detected problem and its cause. Clicking on an alert icon in the program causes the corresponding entry in the table to be highlighted (and vice versa), thereby supporting direct navigation between an alert and its explanation. If the user alters the code to fix a smell, the corresponding icon and table entry automatically disappear.

4.2 Enhanced smell detection with integrated dynamic profiling/hotspot analysis

While the tool described in the above section is useful for helping end-users detect performance problems, the information that it provides is unfiltered and has the possibility of being a false positive.

For example, there may be a Build Array in Loop smell, but if that loop only iterates a few times it likely will have very little impact on the overall performance of the program. Similarly, a Multiple Array Copies smell that involves an array with only 5 elements that is copied only once through a wire split will have a very minor effect on performance. The tool as constructed in the previous section does not have the intelligence to filter these cases out. This may lead to users seeing messages that have little to no effect on the performance and may cause mistrust in the tool.

To help remove the risk of false positives, our protoytpe integrates smell-driven profiling to assess whether each smell instance appears significant enough to bring to the user's attention. Since SDPA can detect smells that cause bad performance, it is likely that most performance problems will be contained in the areas affected by those smells. Smell-driven profiling treats these areas as *potential* hot spots and profiles only those areas. If one of those hot spots, which already contains a smell, is found to be poorly performing, then it is marked and the user is informed of that smell instance. By only profiling the areas that contain smells, the tool limits the adverse affects of profiling (i.e., the massive slowdown in program execution that can occur during profiling) and is still able to determine the areas that are causing performance problems.

The first area of implementation that needed to be completed was to set up a way to profile each smell inside a LabVIEW program. To do this, the idea of timing probes was implemented. LabVIEW currently has the ability to create probes that track data over wires, and this method was modified so that CPU and memory usage could be tracked instead of the data values. While probes are usually created manually by the

users, the timing probes used for smell-driven profiling are automatically and invisibly inserted into the user's program to bracket the region of code containing the smell.

Specifically, when the tool detects a smell, it automatically create two timing probes. The first probe is created on the wire leading into the smell and the second is created on the wire leading out of the smell. If a program is run with smell-driven profiling turned on, it will take a snapshot of current memory usage as well as mark the current time when the first timing probe is hit (during execution). When the second timing probe is hit, it takes another snapshot of the memory usage and the current time. It then subtracts the memory usage and time that was gathered by the first timing probe from the data of the second. This gives the change in memory as well as the total execution time of the area containing the smell.

With the timing probes in place, the profiling of the program is fairly straightforward. LabVIEW provides the ability to take snapshots of the memory usage programmatically and the time at a given point can be gathered with one line of code. When profiling is selected, an initial snapshot of the program will be taken, providing the current level of memory use and the start time. This allows smell-driven profiling to determine the total memory usage and execution of the program by comparing it to a similar snapshot at the end of execution. During execution, if a timing probe is hit a snapshot of memory is taken and the current time is recorded. When the corresponding ending timing probe is hit, another snapshot will be taken gathering the new memory information and current time. The tool compares the values to determine the change in memory usage and the execution time of the profiled area. If the timing probes are hit multiple times then the information for each execution will be averaged to see if the memory usage or execution time increases for each specific run.

The final step of the profiler is to use the timing and memory information gathered from the timing probes to determine which smells to filter out. The heuristic is based on the notion that no smell instance should consume more than a disproportionate share of the overall execution time. For example, if a smell instance occupies 10% of a program's total nodes, and it takes up 90% of the execution time, then it would not be filtered—but if it took up 95% of the program's nodes, then using 90% of the execution time would be reasonable, and the smell instance would need to be filtered.

In particular, the tool computes the percent of the program's nodes occupied by the smell instance (NP), computes the precent of total memory consumed in the smell instance (MP), and computes the percent of total execution time consumed in the smell instance (TP). Currently, based on configurable thresholds, the tool filters out and ignores any smell instances for which TP < NP+15% and MP < NP + 10%. Thus, if the TP and MP are both small, relative to NP, then they are deemed not worth bringing to the user's attention.

5. Evaluation of utility

We performed a laboratory study to evaluate the utility of the static SDPA technique implemented in our prototype, as well as to uncover opportunities for future prototype enhancements. During this within-subjects study, 13 people performed tasks with our tool, and the same 13 people performed comparable tasks without the tool. The tasks and tools were counterbalanced to cancel any learning- and task-related effects.

5.1 Methodology

Participants: We recruited participants by sending emails to Application Engineers at National Instruments. (The study was conducted in National Instruments headquarters.) Participants were required to have at least a year of LabVIEW programming experience, with the intention of ensuring that participants would have a plausible chance of finding and diagnosing performance problems even without our prototype's help.

Tasks: We gave each person two program-troubleshooting tasks, each involving a different LabVIEW program. Every participant performed both tasks. We told participants that each program had one performance issue, and we asked them to verbally identify the problem as well as its solution; actually implementing the fix was not required. We gave a short tutorial about the prototype, explaining what it was designed to do and what information would be displayed. Participants had a total of 40 minutes to do both tasks.

Each participant had access to our prototype tool (SDPA without profiling) for only one of the two tasks. In other words, people performed one task with our tool, and the same people performed the other task without the tool. This within-subject design thus controlled for any between-participant skill differences. In addition, we randomized the assignment of tools to tasks, as well as the ordering of task and tool assignment.

This "counterbalancing" of program ordering and condition ordering (with/without prototype) is a standard empirical method used to control for any between-program differences in difficulty and any between-task learning effects.

We created the two programs for this study based on real-world examples of LabVIEW programs with performance problems that users had asked for help with troubleshooting in the LabVIEW online forum. In order to increase the likelihood that our participants would complete tasks in the allotted time, we modified these posted programs so the effects of the performance problem were more pronounced in the two programs that we actually used for our study tasks. Specifically, these two programs as posted had demonstrated the Build Array in Loop and Multiple Array Copies smells, respectively, which eventually caused very slow execution within a few minutes; we increased the array size in each program so that the problem occurred within seconds.

We do not believe that altering the real-world programs in this fashion biased the study in favor of our prototype because our alterations did not affect the behavior of our smell detectors. If anything, our alterations made it easier for participants to complete tasks *without* the use of our prototype, because the performance problems became more obvious.

Quantitative evaluation: As participants attempted to find and verbally report each problem and its solution, we manually recorded four measures of success: whether participants found each problem, how long finding the problem took, whether they found the solution after finding the problem, and how long finding the solution required. Each program had one problem, for which there was one clearly correct solution (which forum posts confirmed). When participants gave incorrect answers (e.g., stating that the problem was in a place other than the real problem), we told them to keep looking. The participants could give up and move on, if desired (e.g., due to frustration).

We analyzed these measures statistically. We used Fisher's exact test to test two null hypotheses about success (H1 and H2, below). We used the Mann-Whitney U test to test two null hypotheses about time taken by participants (H3 and H4, below). We chose the Mann-Whitney U test instead of a t-test because we did not expect (and in fact did not find) that time taken by participants would be normally distributed.

Null hypotheses, tested with one-tail statistical tests:

H1: Participants will be as likely to find performance problems without the tool as with the tool.

H2: Participants who find performance problems will be as likely to find solutions without the tool as with the tool.

H3: Participants who find a performance problem will do so at least as quickly without the tool as with the tool.

H4: Participants who find a solution (after finding a problem) will do so at least as quickly without the tool as with the tool.

Qualitative evaluation: Subsequent to the tasks, we interviewed participants about their experience with the prototype. In prior studies where we asked quantitative Likert-style questions ("rate this tool's usefulness"), our participants always gave a "helpful" or "very helpful" rating—which was of little use in isolating where tools needed improvement. Therefore, for the current study, we instead used qualitative questions that attempted to draw out *explanations* of what was good or bad about our prototype. For example, we asked "Did the tool ever intrude on your train of thought or distract you from the task at hand? If so, what was distracting?" Further questions asked for specific feedback about the alerts, the frequency of notifications, the kind of the alerts and advice given, the effectiveness at drawing attention to the performance problem, and the prototype's usefulness overall. We analyzed these answers, as in the formative study, with thematic analysis followed by another researcher's inspection of all categorization results to assess face validity.

5.2 Results from the evaluation of utility

Overall, the study showed SDPA had a large impact on participants' ability to diagnose problems.

As shown in the first row of Table 2, all 13 of our study's participants (100%) found the problem when performing tasks while using our tool. However, of these 13 participants, only 5 of them (38%) found the problem when performing tasks when they were without the tool. This difference in success rates was statistically significant ($p < 0.001$).

In the situations where participants were able to find the problem, they took only 4.49 minutes with our tool and 6.85 without, although this difference was not statistically significant (at $p < 0.05$), as shown in the second row of Table 2. In these situations, participants found the solution in 92% of the tasks with the tool and only 60% without (not significant at $p < 0.05$), as shown in the third row of the table.

Finally, as shown in the last row of the table, when they did successfully describe a solution, they were able to do so faster with our tool, taking only 0.83 minutes compared to 2.37 minutes, a statistically significant difference (Mann-Whitney U=32, $p < 0.001$).

There were no significant differences between tasks. In short, participants were more likely to find problems with our tool, and having done so, they were faster at finding solutions.

In response to our interview, 12 of 13 participants mentioned at least one reason why the tool was useful:

- 8 said it helped them *focus* on problem areas. Of these, 3 specifically mentioned the tool alerted them to areas that they had not even considered to be potentially problematic.
- 2 stated that it was a quick *reminder of problems* that LabVIEW programs can have
- 2 stated that the tool's *advice* for improving the program was the useful part.
- 2 felt the tool would help people write *better* code.
- 1 commented that the tool might save people *time*
- 1 said it was a *reminder of good practices* for structuring programs that he already knew from prior training

Taken together, this qualitative feedback strongly indicates that our approach helps people focus their attention, that it might be useful for reminding some users about training issues, and that it might be useful for helping some people create better programs faster.

In response to our questions probing for prototype weaknesses, participants suggested two areas for improvement. First, 4 participants said that more assistance in fixing the performance problem would be helpful. They stated that they understood where the issue was, but the suggestion provided was not enough for them to easily determine a solution. One of these participants suggested providing links to white papers or other documentation to explain how to fix the problem. Second, 8 participants mentioned wanting a way to opt in or opt out of some or all notifications, depending on whether they were creating a program for which performance was important. One cautioned that it would become a "hindrance" if the tool started popping up messages like Microsoft Clippy.

6. Evaluation of accuracy

To assess how well the approach would accurately identify performance problems in general, a corpus of real world LabVIEW programs was gathered that the tool could be run on. The source for these programs was the LabVIEW online forums (http://forums.ni.com/t5/LabVIEW/bd-p/170), where users post questions about how to create or improve programs. While there are many LabVIEW programs that are available on the forums, the parameters of this study required a focused search for programs that had performance problems. Investigating whether each putative performance problem caused a non-trivial impact on CPU and memory required, in particular, assessing whether removing the smell would actually improve performance. Therefore, a corpus of "Before" and "After" programs was needed, namely, the study required a set of pairs of programs, where the first in each pair was an uploaded program from a user that contained a performance problem before fixing, and the second was a version after fixing by an expert so the performance problem had been removed.

6.1 Methodology

To gather this corpus, the LabVIEW forum was searched for specific keywords (specifically, "slow program" and "poor performance"). The results were filtered to only include forum posts that contained attachments. This set of posts was then read to find those that had the original poorly performing LabVIEW code as well as a solution that had been provided by an expert. In this case, an expert was considered to be a person with a high number of posts, a high proportion of posts that had been marked as correct solutions by other users, or a high level of activity on the forums. In some instances, the solution was accepted if it was

provided by a non-expert who did not meet these criteria, but only if an expert subsequently stated in the forum thread that the non-expert's solution was correct and that is what they would have done, or if such an expert made a small suggestion about improving the non-expert's solution but did not upload new code. In these latter cases, if an expert stated that a modification should be made to the program without uploading a solution, then the change was performed manually by the researchers conducting this study, in order to reflect the change that the expert would have made. Forum posts were reviewed until 30 test programs with Before and After versions were obtained.

The section below discusses analyses performed with this corpus to investigate the following four specific hypotheses:

H1: Smells occur in most of the end-user dataflow programs where users report performance problems.

H2: SDPA automatically identifies most performance problems that experts find when fixing performance problems.

H3: Smells identified by SDPA usually indicate non-trivial performance problems.

H4: Smell-driven profiling improves the false positive rate of SDPA and negligibly impacts the true positive rate.

6.2 Results from the evaluation of accuracy

6.2.1 Prevalence of Performance Smells

The first analysis examined the frequency of each smell in order to evaluate whether smells are common in end-user dataflow programs that have performance problems. The first step was to run the SDPA prototype tool on every Before program contained in the repository. This created a list with frequency data for each smell including how often they occurred, both in terms of total number of instances as well as how many unique programs they were contained in. A few of the programs in the repository had subVIs associated with them. In those instances, the data for the subVIs are included in the information for the main program VI and does not show up as having a smell in two separate VIs, to avoid double-counting.

The analysis identified 150 smell instances, or an average of 5 smell instances per program. We found that every Before program in the corpus had at least one smell. These results confirm that smells are common in end-user dataflow programs that have performance problems.

Table 3 shows the information that was gathered for each specific smell, including total number of instances and the number of programs affected. It is clear that not all smells are equally common. The first three smells listed below account for 70% of all the smell instances that occurred in these 30 programs, and the top 5 accounted for 87% of all the smell instances. These statistics confirm our hypothesis that smells occur in most of the end-user dataflow programs where users report performance problems.

6.2.2 Effectiveness of SDPA at Finding Problems Fixed by Experts

To test whether SDPA automatically identifies most performance problems that experts find, all of the Before programs created by users were reviewed and compared to the After programs created by experts. A list was made of all the locations where the Before programs were modified. Then, this list of locations was compared to the list of smell instances that SDPA had identified for the analysis above. It was then possible to compute what proportion of the expert-identified locations were identified by SDPA.

We found that experts made changes in only 8 locations other than the 150 smells identified by SDPA. Thus, SDPA successfully identified the vast majority of the performance problems identified by experts.

The main optimization that was performed by experts that is not caught or even noted by the tool is algorithm refactoring. There were several After programs in the corpus that were completely changed by the expert to make the code more efficient. One example of this was a program that had several nested loops. The expert stripped away all of the extra loops along with the necessary changes to insure the program still created the desired output. This not only fixed the performance problem the user was talking about, but it also made the program much easier to read and understand.

Another important optimization that the experts looked for that was not caught by the tool was to show all of the buffer allocations of a program and to try and minimize them. This is similar in nature to program refactoring, but since it had the specific goal of removing one thing it warrants a discussion. Buffer allocations are particularly useful when doing memory optimizations as it shows dots every place that the LabVIEW compiler made a buffer allocation. This usually means that the program is creating a copy of the value on the wire, and in the case of arrays, this can potentially cause high memory use, particularly when the buffer allocations occur inside loops. While some of the smells that the tool detects can cause buffer

allocations, namely Build Array in Loop, the tool does not check explicitly check for buffer allocations. There were two programs in the corpus where the expert worked to remove as many buffer allocations as possible, which significantly improved performance.

6.2.3True and False Positives of SDPA

Just because SPDA indicated the presence of a smell in a certain location, or just because an expert actually did perform an edit in that location, does not necessarily imply that the smells actually caused real performance problems. Therefore, a further analysis was required to assess what proportion of the smell instances truly harmed performance.

This was accomplished in several steps. First, for each smell identified by SDPA in a Before program, a corresponding fix was created. This fix was taken from the After program if the expert had supplied a fix at that location in the program. Otherwise, a fix was generated using the transformation offered by the tool if applicable. Otherwise, for a few situations where the transformation offered by the tool could not be applied, then the Before version was used unaltered as the fixed version (implying that there was no good fix for the smell, and no performance improvement was possible). Second, the Before program was run and the fixed program was then run. In each case, the execution time and the memory usage were both measured. Third, the percent improvement between the fixed and the original program was noted, both in terms of impact on execution time and on memory usage. Fourth, a smell was categorized as a true positive if its impact was at least 15% on execution time or 10% on memory usage, or a false positive otherwise. Fifth, the overall numbers of true and false positives were computed.

Overall, this analysis revealed that most smells identified by SDPA did indicate an actual performance problem. Of the 150 smell instances, only 48 (32%) were false positives, while 102 (68%) were true positives.

Most of the false positives were due to the Terminals in a Structure smell. While this smell occurs frequently and does cause performance issues, there are cases where a terminal has to be inside of a structure due to the algorithm design. In such a case, as noted above, neither the expert nor the tool could actually fix the problem, so the Before version of the program was used as the fixed form as well, leading to a 0% improvement. In addition, in some cases when terminals could be moved outside of loops, the impact still fell below the threshold to determine a false positive. This does not necessarily signal that this smell should be abandoned, as there are many cases where moving the terminals outside of a structure significantly improves performance.

Another interesting note is that every instance of a sequence structure was determined to be a false positive, as it had little impact on the actual performance of the program. Similarly, the Too Many Variables smell showed little impact on the programs in the corpus, and all but one instance was marked as a false positive.

One interesting set of false positives was a few instances of the Build Array in Loop smell. In these cases, either the Build Array node was in a loop that iterated very few times and did not have an impact on the memory, or it was in a conditional statement that executed very few times inside of the loop. While this smell in considered one of the worst offenders, as noted by AE Specialists in the formative study, the results of this analysis shows that there are certainly cases where it has little negative impact.

6.2.4 True and False Positives with SPDA and Hotspot Profiling

The analysis above was continued by turning on Hotspot profiling, to evaluate Claim 4 that smell-driven profiling improves the false positive rate of SDPA and negligibly impacts the true positive rate. As before, SDPA was run on all of the Before programs to identify smell instances, but smell-driven profiling was also activated in order to provide the execution data needed to filter down the list of smell instances to only include those that also corresponded to Hotspots. This reduced the number of instances identified, which was expected to decrease the number of true and false positives. The question, of course, is whether smell-driven profiling would successfully decrease the false positive rate much more than it decreased the true positive rate.

Enabling smell-driven profiling yielded a total of 112 smell instances, of which 24 were false positives and 88 were true positives. Thus, enabling profiling decreased the total number of false positives by 50% while only decreasing the number of true positives by 13%. Table 4 shows the results obtained.

The primary reason there was not a larger decline in the false positive rate is due to how the smells are profiled. If there are multiple smells in a structure, they are all profiled together by profiling the runtime within that structure. So if a loop has a terminal inside it that is not causing any performance issues, but it

also has a Build Array node that is, both would be marked as a hotspot. This would enable the Terminals in Structure smell to slip through, even though it is not specifically causing problems. How smells are profiled could be refined in future versions by adding profiling inside of structures to determine the exact areas that are causing problems.

The primary reason for the decrease in the number of true positives is that the profiling was unable to measure the impact of Uninitialized Shift Registers as the impacts of that smell only show up when the program is run several times in a row, which the profiler does not check for. For example, one execution of the program which is profiled might use 5000 KB of memory for a 2 minute run. This is not extreme and no significant performance problems are detected. However, with the shift registers uninitialized, every single execution will cause the memory use to increase by around 5000 KB (depending on the run time). The second run will use 10,000 KB, the third 15,000. The profiler does not filter over multiple runs, and if it takes a long time for the problem to show it has some problems determining that it is actually an issue. This problem could be addressed by allowing the profiler to aggregate over multiple runs, thereby enabling it to accumulate enough information to note that a certain smell is actually turning out to be a hotspot.

7. Threats to validity

The primary threat to validity is if the participants in our evaluation were atypical of most LabVIEW programmers, or if our programming tasks were atypical of real-world tasks. We intentionally recruited from a sample frame of people (AEs) who were likely to be able to find and solve performance problems on their own, without our tool. Yet even they found the tool helpful. So we anticipate that typical LabVIEW programmers would benefit even more from our tool than these participants did. Likewise, the programs in our tasks were based on real-world problems, and we adjusted these programs so finding problems without our tool would be easier, which again means that programmers working on typical tasks should benefit from our tool even more than our participants did.

In terms of construct validity, we did not require participants to actually fix performance problems, so though we can conclude that our prototype helps people to diagnose problems, we cannot conclude that it helps them actually fix problems faster. In the qualitative analysis, the codes assigned during thematic analysis were all checked (rather than a sample of code assignments, as in many other studies), and 100% appeared to have face validity. There was little judgment required in this coding because participants' comments were generally quite direct, so the perfect agreement between researchers was unsurprising.

Concerning external validity, our assessment of the tool's accuracy on programs sampled from the online forums helps to establish generalizability across the range of problems that users are likely to encounter in practice. However, we do not claim that the specific smells used in our LabVIEW prototype will generalize to other languages. Nonetheless, we anticipate the overall Smell-driven performance analysis technique would generalize to other visual dataflow languages such as Prograph or Yahoo! Pipes. Such languages and their execution environments typically provide the key properties upon which our technique depends: a graph structure that can be analyzed against heuristics, a visual representation of the program offering places for inserting visual alerts, and visual space for displaying explanations and advice. We also anticipate that our method for identifying problems (i.e., a formative study to interview experts) would generalize to other languages.

8. Related work

Our technique fits within the well-established area of tool support for *software performance engineering* [28], which includes *performance testing and optimization*. The novelty of our work is in its orientation toward the needs of end-user programmers, with a particular emphasis on providing *situated explanations within a visual dataflow language*. We grounded our technique in empirical formative research that led to our technique for identifying bad smells through static analysis. Other tools exist for software performance engineering, but our technique is the first to provide a solution for *how to detect and explain performance problems in a visual dataflow language*.

The most widespread tools for performance analysis are simple execution profilers based on trace logging. These tools record operations during execution and provide a visualization of where a program spends the most time. For example, as shown in Figure 2, the LabVIEW Desktop Execution Trace Toolkit dumps a list of these operations. Other tools illustrate better ways to visualize these data. For example, JumpShot shows a diagram similar to a Gantt chart to reveal how parts of a parallel program depend on one another and how long each part takes [29]. ParaGraph provides similar visualizations [9]. Senseo computes

total time spent on each line of Java code and shows a heatmap alongside the code to highlight areas that use large amounts of resources, such as memory or CPU [21]. The key limitation of approaches like these, particularly from the standpoint of an end-user programmer, is that they incorporate no judgments about performance; these tools simply describe what a program does, not whether a particular part of a program should be able to do better. Instead, programmers must use their own judgment and expertise to recognize and solve problems. Our formative study showed end-user programmers may lack this expertise.

More sophisticated tools use runtime data to help datacenter operators to detect, understand, and respond to performance crises (e.g., [1][18][25][30]). For a small cluster, the operator might be the scientist or programmer; or the operator could be a system administrator hired by the scientist. Tools for this context are important because an increasing trend in scientific programming is the reliance on grid or multicore computing centers running parallelized programs written in traditional textual languages such as Fortran or Java [2]. Powerful tools are also capable of analyzing runtime data collected remotely from millions' of users computers, in order to pinpoint performance problems [7]. Runtime analysis tools work by recording logs of the system's operation, grouping execution traces reconstructed from these logs based on similarity, and detecting problems by identifying aberrant traces at runtime that are highly dissimilar from normal operation. Traces can be analyzed at different levels of granularity to detect node-level problems (e.g., [25]), problems related to a portion of a stack trace (e.g., [7]), or system-wide problems (e.g., [1][18][30]). Tools further categorize these aberrations and, for each category, advise the datacenter operator and/or professional programmers how to fix the problem. This approach could hypothetically be applied to any program regardless of the programming language (though at present it has only been tested with general purpose programming languages like Java). However, unlike static analyses like ours, dynamic analyses provide no design-time guidance (and, in particular, no design-time guidance situated within dataflow source code).

This leaves static analysis approaches, which have been widely applied to help programmers find performance problems in textual code. For example, one early such tool, LISP-CRITIC, analyzed code to generate suggestions of how to improve code [4]. For instance, this tool could identify situations where compound invocations of multiple functions could be replaced with single calls to more powerful functions, thereby improving efficiency. A more modern tool such as Java Critiquer provides similar functionality for Java and includes convenient tools enabling expert programmers to easily create new rules for identifying inefficient code [19]. Approaches such as these are dependent on text-based analysis of code, typically through regular expressions or similar textual patterns. As such, they are inapplicable to visual languages.

Only a handful of tools, like ours, perform a static analysis of dataflow code to suggest places where an end-user programmer could improve performance. Two of these are for a programming environment called Yahoo! Pipes wherein users create programs that operate on newsfeeds; each tool for this environment can identify certain opportunities to eliminate or reorder specific operations to improve performance [8][14]. While these two tools are early prototypes that have not yet been tested with users, National Instruments has provided a much more developed tool called VI Analyzer as an "add-on" for LabVIEW. By add-on, we mean that this tool (like the Desktop Execution Trace Toolkit in Figure 2) is separate from the main IDE. After a programmer creates a program and saves it to a file, the VI Analyzer can then be started and used to analyze the program. Each of these analyses looks for issues that could be problematic from the standpoint of maintainability, coding style, and performance. Figure 3 shows example output.

National Instruments has told us two key limitations restrict VI Analyzer's usefulness in practice. First, because it is separate from the primary IDE, requiring navigation between the two, users must map between the list of problems and the code implicated in those problems. Hard mental operations such as these are well-known to present a barrier to programmers [6]. Second, VI Analyzer currently includes only five tests related to performance, and these focus mainly on program settings rather than the code itself. For example, one of the tests checks to see if debugging is enabled, as this often slows down execution speed. In contrast, our technique of Smell-driven performance analysis detects a broad range of performance problems and displays alerts as well as explanations directly within the editor.

9. Conclusions and future work

In this paper, we have presented the technique of *Smell-driven performance analysis*, which helps end-user programmers to discover and solve performance problems in the source code of a visual dataflow language. The approach accomplishes this by providing situated explanations within the visual dataflow language's IDE. We have developed a prototype implementation for LabVIEW and integrated it with the existing LabVIEW IDE.

We performed a formative study that confirmed our first hypothesis that many performance problems are primarily the result of how end-user programmers implement their visual dataflow programs. The study also led to heuristics we incorporated into our prototype.

In addition, we performed a user study that confirmed our second hypothesis: alerting and advising end-user programmers aids them in finding and understanding how to fix performance problems in visual dataflow code. With our tool, participants successfully identified problems 100% of the time and found solutions 92% of the time; without the tool, these rates fell to 38% and 60%, respectively (Table 2). With the tool, they took less time to find a solution than without the tool. They also said our tool would help people write better code, might save time, and served as a reminder of training that they had already received.

Finally, we performed a study that investigated the accuracy of SDPA and SDPA with profiling on a sample of programs for which users asked help with fixing performance problems. We found that 100% of these programs did contain at least one smell instance, that SDPA correctly identified 150 out of 158 places in the code where expert programmers modified these programs to fix performance problems, that the majority (68%) of locations where SDPA identified a smell instance did correspond to an actual non-trivial performance problem, and that enhancing SDPA with profiling cut the number of false positives by half.

Based on these results, we conclude that *Smell-driven performance analysis* provides a reliable basis for helping end-user programmers to troubleshoot performance problems. Future work will extend this approach in several ways.

First, although our empirical evaluation revealed that the technique as it stands helps to reduce time for finding solutions to performance problems, we plan to further reduce the need for programmer effort by integrating additional techniques for semi-automatic elimination of bad smell instances. We believe that several of the smells could be fixed with transformations that improve performance but leave program semantics *nearly* unaltered; so a tool could quickly fix these smells, with user oversight. We anticipate that other smells could be fixed via a more interactive technique where a tool walks the user through a decision process about how to reorganize the dataflow graph to improve its structure. In the end, we expect that this line of work will ultimately enable end-user programmers to find and fix performance problems in visual dataflow code, rather than having to resort to traditional textual languages.

Second, we are particularly excited that extensions of our technique might be useful for helping to train end-user programmers, given the interest that experimental participants showed in links to white papers about performance problems, as well as their comments about how it helped to remind them of earlier training. Given the fact that our approach provides situated explanations within the IDE, extending it with explicit support for training could make it possible to aid situated learning by end-user programmers within the IDE. Specifically, each smell instance represents a specific case of a general principle for how to improve performance. Prior research has shown that pedagogical materials can be made more effective if they integrate specific examples with more general explanations that help end-user programmers to understand abstractions of which the examples are instances [3]. This contrasts with current LabVIEW training materials, which are separated from the concrete programs with which the programmers are struggling. Future versions of our tool could potentially enhance learning by (1) detecting a smell instance, (2) retrieving a relevant template that explains the associated smell in general, and (3) automatically generating training materials by filling in the template with the details of this particular smell instance. Future studies could assess learning by tracking users over time (perhaps one or more months) to see if the use of known bad smells decreases over time after exposure to the information provided by the tool.

Finally, while the main goal of this research was to find performance problems in an individual's code, there is no reason why it could not be extended to find performance problems in other code available to LabVIEW users. In particular, if a user wants to reuse a LabVIEW program, or a snippet, they could first check it for smells to determine if it should be added. This could be used as a guide to help users find code that would perform well if reused. In addition, the tool could be applied to any large snippet that is pasted or imported. It could quickly be run to notify the user if they have added (or pasted) code that will likely perform poorly.

This idea could easily be extended and applied to an entire repository of LabVIEW programs. When the programs are uploaded to be shared they could be checked by the tool for any smells that might exist. This would help to determine the quality of a given program and help users pick programs that would not perform poorly. Currently, there are no quality-filtered online repositories that LabVIEW users can access, so creating one could ease reuse and help to reduce the effort and frustration that users may encounter when attempting to reuse existing code.

ACKNOWLEDGMENT

We thank National Instruments for funding this research, helping to recruit study participants, and providing access to the latest version of the LabVIEW development environment. 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.

REFERENCES

[1] Bodik, P., Goldszmidt, M., Fox, A., Woodard, D., and Andersen, H. (2010) Fingerprinting the datacenter: Automated classification of performance crises. *5th ACM European Conf. on Computer Sys.*, 111-124.

[2] Carver, J., Kendall, R., Squires, S., and Post, D. (2007) Software development environments for scientific and engineering software: A series of case studies. *29th ACM/IEEE Intl. Conf. on Software Engineering (ICSE)*, 550-559.

[3] Dorn, B. (2011) ScriptABLE: Supporting informal learning with cases. *Proceedings of the Seventh International Workshop on Computing Education Research*, 69-76.

[4] Fischer, G. (1987) A critic for LISP. *10th Intl. Joint Conf. on Artificial Intelligence*, 77-184.

[5] Fowler, M., and Beck, K. (1999) *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional.

[6] Green, T., and Petre, M. (1996) Usability analysis of visual programming environments: A 'Cognitive dimensions' framework. *J. of Visual Languages and Computing*, *7*(2), 131-174.

[7] Han, S., Dang, Y., Ge, S., Zhang, D., and Xie, T. (2012) Performance debugging in the large via mining millions of stack traces. Proceedings of the 2012 International Conference on Software Engineering, 145-155.

[8] Hassan, O., Ramaswamy, L., and Miller, J. (2010) Enhancing scalability and performance of mashups through merging and operator reordering. *2010 IEEE International Conference on Web Services* (ICWS), 171-178.

[9] Heath, M., and Etheridge, J. (1991) Visualizing the performance of parallel programs. *IEEE Software*, *8*(5), 29-39.

[10] Jones, M., and Scaffidi, C. (2011) Obstacles and opportunities with using visual and domain-specific languages in scientific programming. *IEEE Symp. on Visual Languages and Human-Centric Computing*, 9-16.

[11] Katagiri, H., Furukawa, K., and Anami, S. (1999) RF monitoring system in the Injector Linac. *Intl. Conf. on Accelerator and Large Experimental Physics Control Sys.*, 69-71.

[12] Kerry, E., and Snyder, D. (2010) Comparing LabVIEW graphical code to text-based alternatives for use in test applications. *IEEE AUTOTESTCON*, 1-2.

[13] Krippendorff, K. (2012) *Content Analysis: An Introduction to Its Methodology*, Sage Publications.

[14] Liu, J., Wei, J., Ye, D., and Huang, T. (2010) A new approach to performance optimization of mashups via data flow refactoring. *Proceedings of the Second Asia-Pacific Symposium on Internetware*, 6:1-6:8.

[15] Myers, B., Hudson, S., and Pausch, R. (2000) Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, *7*(1), 3-28.

[16] Nardi, B. (1993) *A Small Matter of Programming: Perspectives on End User Computing*, MIT Press.

[17] National Instruments. (2010) *The LabVIEW Compiler - Under the Hood*. [Online]. http://zone.ni.com/devzone/cda/pub/p/id/1177#toc1 (Last Accessed: February 4, 2013)

[18] Parsons, T., and Murphy, J. (2008) Detecting performance antipatterns in component based enterprise systems. *J. of Object Technology*, *7*(3), 55-90.

[19] Qiu, L., and Riesbeck, C. (2004) Making critiquing practical: Incremental development of educational critiquing systems. *ACM Intl. Conf. on Intelligent User Interfaces (IUI)*, 304-306.

[20] Repenning, A., and Ioannidou, A. (2006) AgentCubes: Raising the ceiling of end-user development in education through incremental 3D. *IEEE Symp. on Visual Languages and Human-Centric Computing*, 27-34.

[21] Rothlisberger, D., Harry, M., Villazon, A., Ansaloni, D., Binder, W., Nierstrasz, O., and Moret, P. (2009) Augmenting static source views in IDEs with dynamic metrics. *IEEE Intl. Conf. on Software Maintenance (ICSM)*, 253-262.

[22] Sanders, R., and Kelly, D. (2008) Dealing with risk in scientific software development. *IEEE Software*, *25*(4), 21-28.

[23] Smedley, T. (1992) Using pictorial and object oriented programming for computer algebra. *ACM Symp. on Applied Computing*, 1243-1247.

[24] Subrahmaniyan, N., Beckwith, L., Grigoreanu, V., Burnett, M., Wiedenbeck, S., Narayanan, V., Bucht, K., and Drummond, R. (2008) Testing vs. code inspection vs. what else?: Male and female end users' debugging strategies. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 617-626.

[25] Tan, J., Pan, X., Marinelli, E., Kavulya, S., Gandhi, R., and Narasimhan, P. (2010) Kahuna: Problem diagnosis for mapreduce-based cloud computing environments. *IEEE Network Operations and Management Symp.* (NOMS), 112-119.

[26] Whitley, K., and Blackwell, A. (2001) Visual programming in the wild: A survey of labVIEW programmers. *Journal of Visual Languages & Computing*, *12*(4), 435-472.

[27] Wilson, A., Burnett, M., Beckwith, L., Granatir, O., Casburn, L., Cook, C., Durham, M., and Rothermel, G. (2003) Harnessing curiosity to increase correctness in end-user programming. *ACM/SIGCHI Conf. on Human Factors in Computing Sys. (CHI)*, 305-312.

[28] Woodside, M., Franks, G., and Petriu, D. (2007) The future of software performance engineering. *Workshop on the Future of Software Engineering, IEEE Intl. Conf. on Software Engineering (ICSE)*, 171-187.

[29] Wu, C., Bolmarcich, A., Snir, M., Wootton, D., Parpia, F., Chan, A., Lusk, E., and Gropp, W. (2000) From trace generation to visualization: A performance framework for distributed parallel systems. *ACM/IEEE Conf. on Supercomputing*, 50-67.

[30] Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. (2009) Detecting large-scale system problems by mining console logs. ACM *22nd Symp. on Operating Sys. Principles*, 117-132.