# Adding Control-Flow to a Visual Data-Flow Representation

David Dearman
Faculty of Computer Science
Dalhousie University
Halifax, Nova Scotia, Canada
dearman@cs.dal.ca

Anthony Cox
Faculty of Computer Science
Dalhousie University
Halifax, Nova Scotia, Canada
amcox@cs.dal.ca

Maryanne Fisher
Department of Psychology
Saint Mary's University
Halifax, Nova Scotia, Canada
mlfisher@smu.ca

## Abstract

*Previous studies have shown that novices do not tend to extract or use data-flow information during program comprehension. However, for impact analysis and similar tasks, data-flow information is necessary and highly relevant. Visual data-flow programming languages, such as Prograph/CPX, have been commercially successful, suggesting that they provide effective data-flow representations. To explore data-flow representations for program comprehension, we augment Prograph data-flow programs with control-flow features to determine the effects on comprehension. We hypothesize that combined control/data-flow representations will aide comprehension better than data-flow alone. To validate this hypothesis, we present the results of an experiment comparing three combined representations against a data-flow only representation. While the addition of control-flow was found to be beneficial, the complexity of the representations plays an important role. Complex and highly detailed control-flow, although perceived as useful, is less effective when combined with data-flow, than less detailed and less complex control-flow descriptions. This finding suggests a tradeoff between a representation's content and complexity. We found a nested representation describing inter-method control-flow to be the most effective for supporting program comprehension.*

## 1 Introduction

When programmers need access to data-flow information, program comprehension tools must be able to effectively display this information. While data-flow may be represented independently of control-flow, we believe that merging the two flows will improve programmers' ability to perform program comprehension tasks. To validate this hypothesis, we experimentally examined the effects on program comprehension of four data-flow representations based on the visual data-flow language Prograph/CPX [12].

Compared to control-flow, there is less research on the representation of data-flow for supporting program comprehension. It is likely that this lack of research stems from the fact that data-flow information is difficult to extract from procedural programs [11] and that it is rarely extracted by novices [2]. Even when provided with data-flow representations, novices have difficulty using the representation to facilitate comprehension [10].

While there is contradictory evidence as to whether data-flow representations offer advantages over control-flow representations [1, 5], programmers are not always free to choose the most effective representation. Programmer choice is necessarily restricted by the maintenance task they are performing and the information needed to adequately perform the task.

Green [4, 7] has shown that a notation that highlights data-flow leads to better comprehension of data-flow information. Hence, to successfully perform maintenance tasks, such as impact analysis, for which accurate knowledge of data-flow is necessary, programmers benefit from the availability of effective data-flow representations.

Research on visual programming languages [6] has demonstrated that specific properties of visual programming languages (VPLs) lead to differences in program comprehension. When presented with a data-flow VPL, users tend to write program summaries that reference functional information, and when presented with a control-flow VPL, program summaries reference procedural information. As data-flow VPLs, such as Prograph, have experienced some commercial success, we believe that these languages can be used to develop representations for aiding program comprehension.

During requirements analysis, data flow diagrams (DFDs) are often used to describe the data-flow of the application being designed. Freeman [3] documented the problems that occur during DFD creation and the difficulties experienced by analysts when working with DFDs. In particular, analysts often lack the training needed to work with DFDs and, perhaps as a result, have difficulty understanding

data models. Millet [9] suggested that these difficulties are reduced when the DFD is simplified by using a higher level of abstraction for some features. Lloyd and Jankowski [8] explored the application of cognitive information processing principles for adding clarity to DFDs and demonstrated that comprehension is related to DFD clarity. These studies suggest that, even if used for other tasks, data-flow is a difficult concept for programmers and is easily affected by the representation technique used.

As we are unaware of any studies that attempt to determine the optimal representation for data-flow to aide program comprehension, the research in this paper is a first step towards this goal. Mosemann and Wiedenbeck have shown that control-flow views of programs facilitate program comprehension [10]. Thus, we explore the addition of control-flow information to data-flow representations in the belief that this addition will create better representations for use during program comprehension.

To explore the effectiveness of various representation techniques, we experimentally compare the effects on comprehension of four representations based on Prograph/CPX. Each representation is now detailed before the experiment is described and its results discussed.

## 2 Data-Flow Representation

The four representations investigated in this paper are identified as the linear, nested, hierarchical, and augmented representations. The linear representation is identical, except for the method ordering, to the standard Prograph data-flow representation. The other three representations are all created through the addition of control-flow information to the standard Prograph representation. Each representation is now described.

### 2.1 Linear Representation

The linear representation presents each Prograph/CPX method (procedure) in a separate window. The methods are displayed, left to right, on a single line and ordered alphabetically by method name. Figure 1 provides an example of the linear representation. The program in the figure performs a quicksort of a user-entered list of integers.

In Prograph, windows are arranged according to the order of their creation. The switch to alphabetical ordering provides a repeatable ordering for use during experimentation without significantly modifying the representation.

If a method has multiple windows, as occurs when the method has multiple implementations, the two windows are adjacent, with the most complex being the leftmost. Complexity is determined by counting the number of elements in each window and considering the window with the most elements as the most complex. The linear representation is intended to serve as the control case in the experiment, as this representation is not augmented or manipulated in any meaningful way.

### 2.2 Hierarchical Representation

The hierarchical representation extends the linear representation by connecting 'calling' and 'called' methods. As can be seen in Figure 2, every method contains two horizontal bars, one at the top and one at the bottom. The top bar represents the method's incoming values (actual arguments) and the bottom bar represents the method's outgoing (returned) values. In the hierarchical representation, the outgoing bar of a method is aligned and connected, with dotted lines, to the incoming bar of another method when the first method calls the second. Thus, this representation uses vertical alignment to indicate 'calling-called' relationships by placing calling methods higher than called methods. The program in Figure 2 performs the same quicksort as in the linear representation and differs only by the explicit indication of 'calling-called' relationships. The representation gets its name from the hierarchy created by connecting the horizontal bars of the methods. When a method has alternative implementations, they are placed side by side in the hierarchy, with the most complex being the leftmost.

### 2.3 Nested Representation

Similar to the hierarchical representation, the nested representation attempts to make 'calling-called' relationships explicit. However, in this representation nesting is used to indicate the call chain. The nested representation nests methods within each other, according to their calling sequence. Figure 3 shows the same program as Figure 1, but using the nested representation. So that the innermost method is easily viewed, it is kept constant in size, with methods increasing in size as one moves outwards. If a method has two implementations, the most complex is shown and the simpler is layered underneath the more complex. We chose this layering because in Prograph, the simplest implementation frequently corresponds to a base-case condition of a recursive method and the base-case typically contains only minimal information.

Early versions of Prograph supported the use of nesting, for control-flow constructs, during program implementation, but the feature was removed because it required methods to be resized every time method calls were added or deleted.[1] While this limitation prevents nesting from forming the basis of a data-flow representation during program development, the generally static nature of program code during comprehension tasks does not prevent nesting from being effectively employed.

[1] Personal communication with Prograph designer P. Cox.

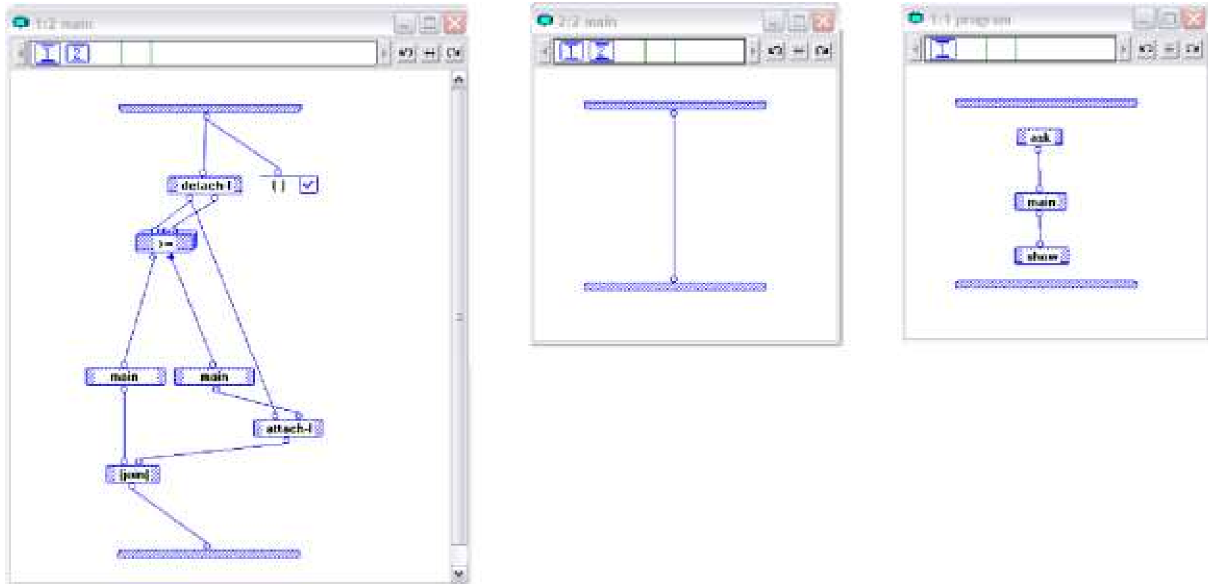


**Figure 1. Example of the Linear Representation**

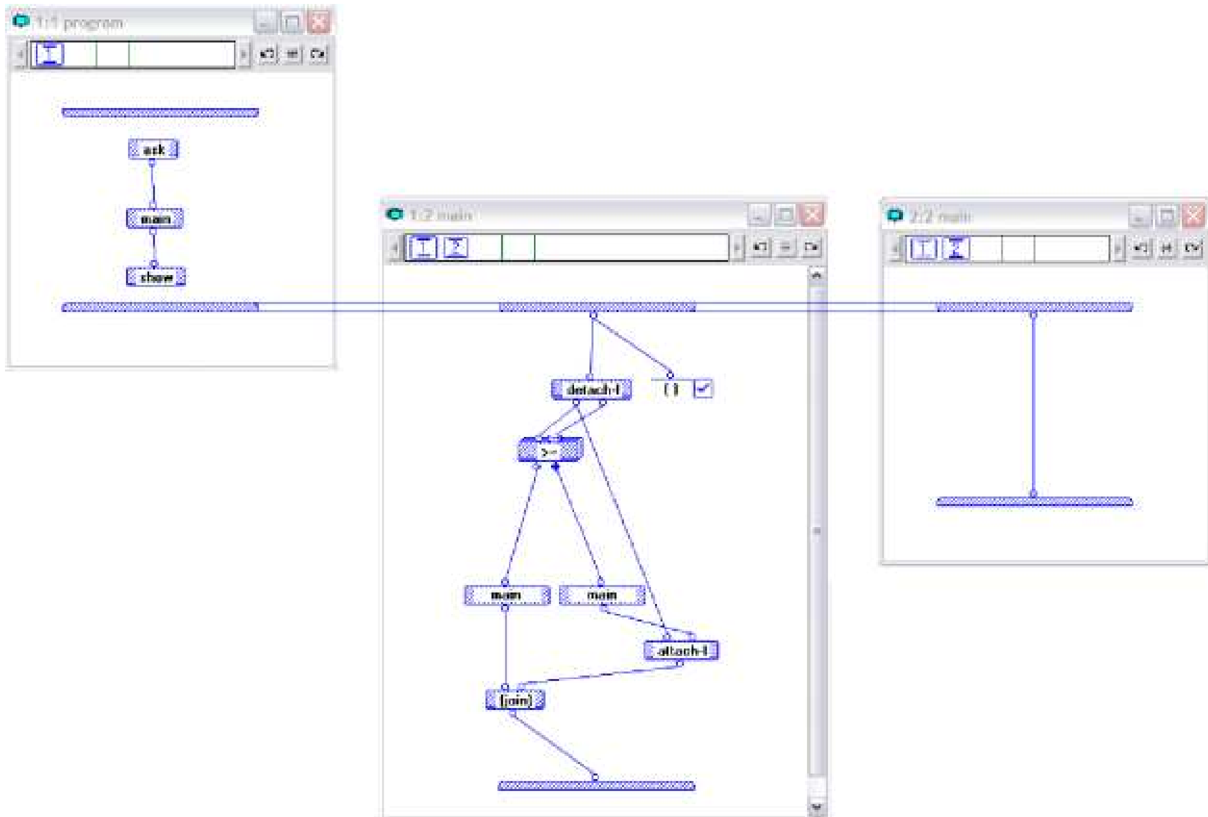


**Figure 2. Example of the Hierarchical Representation**

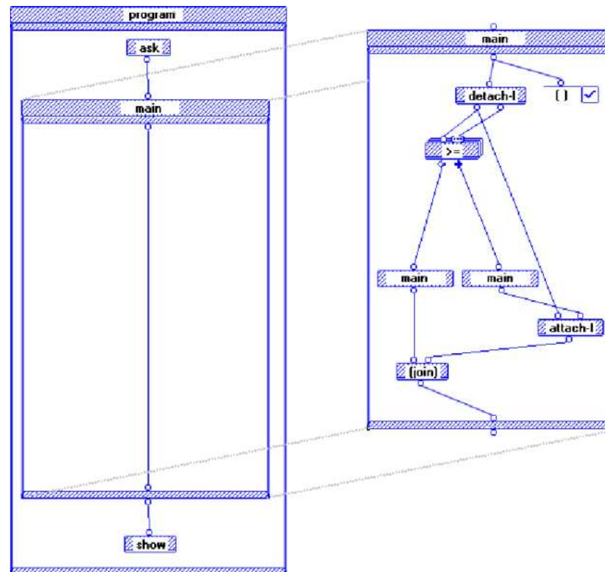


**Figure 3. Example of the Nested Representation**

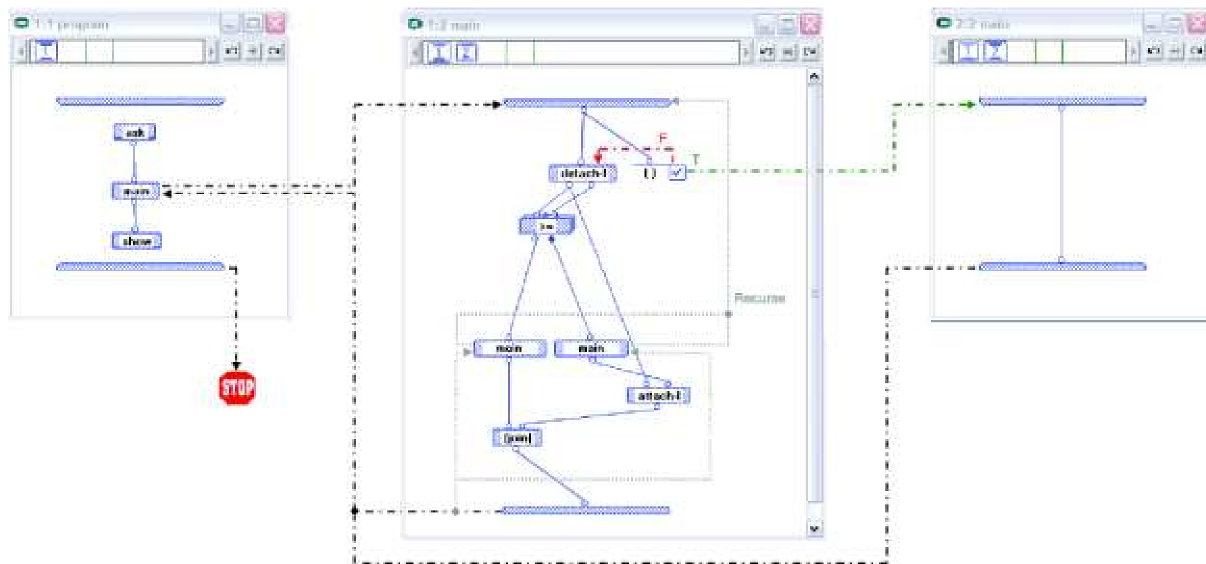


**Figure 4. Example of the Augmented Representation**

### 2.4 Augmented Representation

The augmented representation supplements the basic Prograph/CPX methods with additional control-flow information. In Figure 4, the sample quicksort program is shown using the augmented representation.

Black dashed arrows are used to connect a method call to the horizontal input and output bars of its implementation, and consequently, to indicate that when the method is executed, control is transferred to that method. Conditional elements have green and red arrows (labeled 'T' and 'F') to identify the flow of execution when the condition evaluates to 'true' or to 'false.' A small stop sign is used to mark the exit point of the program. When control-flow deviates from data-flow, soft gray dotted arrows are used to indicate intra-method control flow (loops). If a method is recursive, its input and output bars are connected with a soft gray dotted arrow labeled 'recurse.'

### 2.5 Hypotheses

Apart from the nested representation, the representations all use the same windowing decomposition as the standard Prograph/CPX editor. In the nested representation, the methods are increased in size to allow for nesting and simpler versions of methods (with more than one implementation) are hidden from view. We selected Prograph since, as one of the first successful VPLs, the language is typical of commercial VPLs such as LabVIEW, Softwire and HP VEE. These languages all use similar approaches for representing procedural abstraction.

The linear representation can be considered as equivalent to the standard Prograph representation, while the other three representations all provide some additional control-flow information. The hierarchical and nested representations show inter-method control-flow (calling sequences), and the augmented method shows both inter-method and intra-method control-flow.

The addition of control-flow, to all but the linear representation, was explored since control-flow has been shown to facilitate comprehension when used with textual languages [10]. As well, participants have been found to perform significantly faster when creating program summaries using a control-flow VPL than when using a data-flow VPL [6]. These findings suggest that programmers will benefit when control-flow information is added to a data-flow representation.

The hierarchical and nested representations use window arrangement to explicitly provide additional information without adding a significant amount of material to the screen. The augmented representation adds a considerable amount of material to the screen, but in doing so makes intra-method control-flow very explicit. As programmers are traditionally trained in control-flow management and as it has been shown as the first abstraction developed by novices during comprehension [11], it is likely to be the most useful.

**Hypothesis 1** *The utility of a representation will increase with the amount of added control-flow information. The linear representation will be the least useful, the hierarchical and nested representations will be of intermediate utility and the augmented representation will be the most useful for facilitating comprehension.*

As the representations have varying amounts of additional information, we hypothesize that participants will view the representations with more information as more complex.

**Hypothesis 2** *The linear representation will be rated as the least complex, the hierarchical and nested representations as intermediate in complexity, and the augmented representation as the most complex.*

To test these hypotheses, we performed an experiment to compare the four representations. This experiment is now detailed.

## 3 Experimental Comparison

A paper-based survey was designed to explore how the four data-flow representations affect users' ability to comprehend programs. Participants were shown the four representations and asked to identify each representation, perform comprehension tasks, and indicate the perceived usefulness and complexity of each representation.

Two applications were chosen to test the four representations: a factorial calculator and the quicksort shown in Figures 1 to 4. The factorial calculator was considered to be the simpler of the two applications, as it has fewer nodes (9 as opposed to 10 in the quicksort) and does not use recursion. As well, participants were provided with a tutorial containing an example of Euclid's GCD algorithm. As the factorial calculator was considered to be more like the example than the quicksort, it was potentially easier for participants to understand.

### 3.1 Participants

There was a total of 26 participants in the study; 17 males (age in years, $M = 24.12$, $SD = 3.68$) and 9 females ($M = 26.33$, $SD = 6.58$). All of the participants were either fourth year undergraduate or graduate students in computer science. The use of senior undergraduate and graduate students ensured that all participants were familiar with the concepts of control-flow and data-flow. Eight participants indicated that they possessed previous experience using visual languages, including Prograph, LabVIEW, Toontalk,

and Stagecast Creator; of these eight, five indicated that they had experience using Prograph/CPX.

Participants were drawn from the Faculty of Computer Science at Dalhousie University in Halifax, Canada. The above average experience with visual programming occurs since Philip Cox, the creator of Prograph, teaches a popular course on visual programming at Dalhousie. One participant was excluded from some data analyses as he did not fully complete the survey. Participation was voluntary and participants were not renumerated for their involvement.

### 3.2 Stimuli

The survey consisted of four parts: a demographic questionnaire, an introductory tutorial on Prograph/CPX and two program comprehension tasks. The demographic questionnaire determined the age, gender, educational background and visual language experience of the participants.

The tutorial began with a simple introduction to Prograph/CPX and the representation of data-flow in Prograph. Next, participants were shown a simple implementation of *Euclid's* algorithm to calculate the greatest common divisor of two numbers. The implementation was represented in both C and Prograph. The example was accompanied with a detailed description of its elements and run-time characteristics. The tutorial was meant to ensure that all participants had sufficient knowledge of Prograph so that they could understand the four representations when they were presented during the program comprehension tasks.

The C and Prograph/CPX examples were contrasted so that participants familiar with C would be able to relate the different structural components of a textual language with those of a visual language. In addition, a summary of all Prograph methods that are used in the comprehension tasks was attached to the tutorial. Participants were allowed to retain the tutorial and method summary during the entire study for use as a reference.

Each of the comprehension tasks, which we call 'applications' for simplicity, began with the random placement of the four representations in front of each participant. The first application presented to the participants was the factorial calculator and the second was the quicksort. Each representation was identified using a coloured sticker (red, yellow, green, or blue) and referred to throughout the study by its colour. Colours were kept consistent for both applications, but the order of the representations was varied.

Each application survey first asked participants to identify the content of the four representations. While it was intended that this information be coded and analyzed, the responses varied widely, and thus, prevented the researchers from accurately determining whether participants understood the purpose of the representations. Some participants described the representations in general terms (e.g. "code in flow of execution order") while others described the representations with respect to each application (e.g. "if input 0, next case, else ..."). In many instances the responses were very ambiguous making it difficult to determine whether the participants understood the representations' information content. Examples of ambiguous responses include: "control-flow," "hierarchy," "program decomposition into methods," "layers," "nested methods" and "program structure." These descriptions, while often correct, do not provide sufficient detail to indicate whether participants understood the purpose of the representation. We did not identify the role of the representations since we wanted to determine each representation's ability to present information in an intuitive and easily grasped manner.

Next, participants were asked to rate both the usefulness and complexity of each representation on a scale of 1 to 5. These ratings are referred to as perceived utility and perceived complexity since they are generated by participant perceptions and not by task performance. The utility score labels accompanying the values (from 1 to 5) were: none, lacking, somewhat, useful, and extremely useful. The complexity score labels (from 1 to 5) were: overly simple, simple, good, complex, and overly complex.

Participants were then asked to state the effects of two, application-specific changes. The effect descriptions were scored using a 4 point scale as follows: 1 indicated the description was completely incorrect, 2 indicated partial correctness, 3 indicated a completely correct but incomplete description, and 4 indicated a correct and complete description. After each effect description, participants indicated on a check-list which of the representations they used to determine the effects. The choice of representation was considered as a measure of applied utility and given a score of 0, not used, or 1, used. To ensure that the control-flow information was relevant, some of the changes explicitly affected control-flow. For example, in one of the changes, participants were asked to identify the effect of changing a Boolean expression in the guard of a conditional construct.

Finally, participants were asked to identify the purpose of the program by generating a short, single-line program summary. As the summaries were very short, they were coded using the same 4 point scale as the effect descriptions. A total comprehension score, from 3 to 12, was generated by summing both effect description scores and the program summary score.

### 3.3 Procedure

Participants were seated in a quiet room with a large open table to work on. They were instructed that they were under no time constraints and that they could take as much time as needed to complete the study. The majority of participants completed the study in about an hour.

| Application 1 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Linear | | Hierarchical | | Nested | | Augmented | |
| Measure | M | SD | M | SD | M | SD | M | SD |
| Perceived Utility | 2.31 | 1.12 | 2.35 | 1.16 | 3.65 | 0.89 | 3.73 | 1.15 |
| Complexity | 2.62 | 1.17 | 2.54 | 0.99 | 3.04 | 0.96 | 3.88 | 0.91 |

| Application 2 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Linear | | Hierarchical | | Nested | | Augmented | |
| Measure | M | SD | M | SD | M | SD | M | SD |
| Perceived Utility | 2.08 | 1.00 | 2.04 | 1.18 | 3.12 | 1.21 | 3.96 | 0.98 |
| Complexity | 2.35 | 1.16 | 2.46 | 0.86 | 2.96 | 0.96 | 3.50 | 1.03 |

**Figure 5. Descriptive Statistics**

Participants were first given a consent form, followed by the demographic questionnaire. After completion of the questionnaire, participants were given the Prograph/CPX tutorial and method summary, and instructed to study it and then keep it as a reference for the remainder of the study. Once participants indicated they had reviewed the tutorial and were ready to proceed, the first application was presented. The four different representations were placed in a random left-to-right ordering in front of the participant. Participants were instructed that the ordering of the representations was completely random and they should not consider it as significant. As four representations were being tested, it was not possible to collect a sufficient number of surveys for all possible 24 orderings of the representations. Thus, a randomized approach was used to avoid introducing any ordering effects.

Participants were given scrap paper that they could use to make notes. For the most part, later informal analysis revealed that the scrap paper was used to generate program execution traces for arbitrary inputs. After the representations were distributed, they were given an appropriate survey sheet for each application.

If at any point participants had questions concerning the operation of the Prograph methods, they were directed to the method summary sheet. After participants indicated that they had completed the first application, the representations were removed and the second application was presented. Once again, the four different representations were placed in front of the participants in a random left-to-right ordering, and the participants reminded of this fact. The order of the two applications was consistent so that participants would encounter the easiest task first and not be overwhelmed, as may have happened if the harder task was presented first. Once the participants had indicated completion of the second application, they were debriefed and encouraged to provide feedback.

### 3.4 Results

To minimize the chance of error, all tests are two-tailed and use a level of significance of $\alpha = .05$, unless otherwise stated. We first present the results for the first application before proceeding to the second application.

In the experiment, we asked each participant to rate each representation in terms of its perceived usefulness on a 1–5 scale (useless to useful). As well, participants rated the complexity of each representation on a 1–5 scale (simple to complex). The descriptive statistics for these ratings are presented in Figure 5.

To explore which representations were significantly rated as more useful than others, we performed paired-samples $t$-tests, using the level of significance $\alpha = .01$ to compensate for the number of comparisons. There were no significant differences between augmented and nested $t(25) = .23$, $p = .82$, but the comparisons of augmented and linear $t(25) = 4.75$, $p \leq .001$, and augmented and hierarchical $t(25) = 3.99$, $p \leq .001$ were significant, as the augmented representation was perceived as significantly more useful than the other representations. There was a significant difference between nested and linear, $t(25) = 4.59$, $p \leq .001$, and nested and hierarchical, $t(25) = 4.25$, $p \leq .001$. Finally, the comparison between linear and hierarchical was not significant, $t(25) = .10$, $p = .92$.

Pearson's $r$ correlations were used to examine the relationship between perceived usefulness and comprehension. These correlations were: augmented $r(26) = -.21$, $p = .31$, nested $r(26) = .51$, $p = .008$, linear $r(26) = .29$, $p = .16$, and hierarchical $r(26) = -.022$, $p = .92$. In other words, the nested representation is the only one where perceived usefulness significantly correlates with comprehension.

We then performed correlations to examine the rela-

tionship between applied usefulness (a value from 0 to 2 formed by counting the number of times the representation was used to determine change effects) and comprehension. These Pearson's $r$ correlations revealed no significant relationship between the usefulness of the augmented $r(26) = -.38$, $p = .056$, linear $r(26) = .30$, $p = .14$, or hierarchical $r(26) = .24$, $p = .23$ representations and comprehension. However, there was a significant correlation for the nested representation and comprehension, $r(26) = .44$, $p = .024$.

To test the possibility of a relationship between perceived usefulness and perceived complexity, we compared the two scores. Pearson's $r$ correlations yielded a significant negative correlation between the perceived usefulness and complexity for the augmented representation $r(26) = -.72$, $p \leq .001$. Similarly, the nested representation produced $r(26) = -.41$, $p = .040$. The remaining two representations showed no significant relationship between perceived usefulness and complexity: linear $r(26) = -.08$, $p = .69$, and hierarchical $r(26) = -.099$, $p = .63$. Therefore, as suggested by the strong negative correlation for the augmented and nested representations, there may be a trade-off between usefulness and complexity. The same general pattern was revealed when we correlated complexity and applied usefulness: augmented $r(26) = -.49$, $p = .011$, nested $r(26) = -.37$, $p = .062$, linear $r(26) = .017$, $p = .94$, and hierarchical $r(26) = -.085$, $p = .68$.

It should be noted that the correlation values for the linear and hierarchical representations must be considered with caution for the applied usefulness scores. There were only three individuals in the linear case and two in the hierarchical case who used these representations, yielding a small number of participants with moderate to high scores for applied usefulness.

To confirm the trends found in the first application, the statistics were analogously and independently performed on the data for the second application. However, due to a partially completed survey, the statistics for the second task vary with respect to the number of degrees of freedom. The descriptive statistics for the second application are presented in Figure 5.

Paired samples $t$-tests conducted on these usefulness scores revealed all comparisons were significant excluding the comparison of linear and hierarchical, $t(25) = .22$, $p = .83$. The other comparisons were as follows: augmented and nested $t(24) = 2.63$, $p = .015$, augmented and linear $t(24) = 5.64$, $p \leq .001$, augmented and hierarchical $t(24) = 7.33$, $p \leq .001$, nested and linear $t(25) = 2.90$, $p = .008$, and nested and hierarchical $t(25) = 2.80$, $p = .010$.

We also investigated the relationship between perceived usefulness and comprehension, which revealed the following correlations: augmented $r(25) = -.398$, $p = .049$, nested $r(26) = .23$, $p = .26$, linear $r(25) = .42$, $p = .039$, and hierarchical $r(26) = -.49$, $p = .011$. Thus, only for the nested representation did perceived usefulness not significantly correlate with comprehension.

We next examined applied usefulness and comprehension, again using Pearson's $r$ correlations. The augmented representation yielded $r(26) = -.60$, $p \leq .001$, nested $r(26) = .41$, $p = .037$, linear $r(26) = .39$, $p = .047$, and hierarchical $r(26) = .43$, $p = .029$. Individuals may have perceived the augmented representation to be the most useful (by giving it the highest mean usefulness score), however, there was a strong negative relationship between its applied usefulness and comprehension, such that as usefulness increased, comprehension decreased. The opposite relationship was found for the nested representation, as there was a strong positive correlation, meaning that as usefulness increased, so too did comprehension. The findings for the linear and hierarchical representations are interesting, but as with the first application are the product of very low sample sizes. While participants perceive these representations to have some utility, these perceptions did not result in the representations being used to determine change effects. Thus, we are hesitant to develop meaningful interpretations of the results for the linear and hierarchical representations.

There were no significant correlations between complexity and perceived usefulness: augmented $r(25) = -.23$, $p = .28$, nested $r(26) = -.17$, $p = .41$, linear $r(25) = .13$, $p = .55$, and hierarchical $r(26) = .30$, $p = .14$. Similarly, there were no significant correlations between complexity and applied usefulness: augmented $r(26) = -.18$, $p = .37$, nested $r(26) = -.071$, $p = .73$, linear $r(26) = .10$, $p = .62$, and hierarchical $r(26) = -.061$, $p = .77$.

## 4 Discussion

Analysis of the first application indicates that participants perceived the augmented and nested representations to be of similar utility, as were the linear and hierarchical representations. In the second task, the similarity between the augmented and nested representation diminished and the augmented was perceived as being the most useful. These findings support the hypothesis (H1) that users will perceive a representation to increase in utility as control-flow information is added.

Participants did not perceive the hierarchical representation to have any additional value over the linear representation. It is likely that this lack of difference occurs because participants are unable to determine the purpose of the hierarchy. When asked the purpose of the hierarchical representation, participant comments included: "(it) made no sense at all," "3 separate windows that are somehow re-

lated," "connection between methods, but only a little bit," "nothing really," "don't know," "I don't like this one" and "meaningless connection between methods." From these comments, it appears that the hierarchical representation lacks clarity and does not present information in an intuitive or easily understood manner. The participants did not provide these sorts of comments for any of the other representations. Instead, some participants expressed mild confusion as to the purpose of the linear representation because they were under the impression it contained some additional information not included in the standard Prograph representation of the tutorial.

In both applications, applied usefulness significantly correlates with program comprehension for the nested representation. Thus, there is little doubt that participants find the nested representation to be an effective aide to comprehension. For both applications, the nested representation had a mean complexity close to 3, which had the label 'good.' For the first application, $M = 3.04$ and for the second application, $M = 2.96$. Participants found that the nested representation was neither too complex nor too simple; a feature that likely leads to its ability to improve comprehension.

Although applied utility correlates with comprehension for the augmented representation in the second application, the correlation is negative indicating that the representation harms comprehension. It is likely that this result is a consequence of complexity. The augmented representation has mean complexities of 3.88 and 3.50 for the two applications. Thus, comprehension is potentially impeded because participants are struggling to manage the amount of information contained within the representation.

No generalizations can be drawn for the hierarchical representation since participants had difficulty determining its purpose and consequently very few chose to use it to determine a change effect. The lack of use prevents a meaningful statistical analysis from being performed. As the hierarchical representation contains the same information content as the nested representation, it can be seen that for these sample applications, nesting is a presumably more effective way of presenting intra-method control-flow. Of course, it is possible that an alternative hierarchy, perhaps more tree-like in nature, would be more easily understood by participants and have the same effects on comprehension as the nested representation.

It is interesting that perceived usefulness does not necessarily match applied usefulness. While participants perceive the augmented representation to be useful, use of the representation does not lead to improved comprehension. It is likely that the augmented is perceived as useful because it is either familiar, or because its complexity suggests it contains much useful information. In any case, the measure of perceived usefulness demonstrates the hazards involved in using any subjective measurement; the perception of the participants is not necessarily accurate with respect to their task performance.

Our hypothesis (H1) on the utility of the representations is partly verified. In terms of perceived utility, the hypothesis is found be accurate. The linear representation is the least useful, the nested and hierarchical representations are of intermediate utility and the augmented representation is perceived as the most useful. For no representation does perceived utility consistently correlate with comprehension. In terms of applied utility, the linear and hierarchical representations are not selected frequently enough to be considered useful, the nested representation improves comprehension and the augmented representation hinders comprehension.

For the issue of complexity our hypothesis (H2) is fully verified. Inter-method control-flow is less complex than combined inter/intra-method control-flow, but more complex than a representation with no control-flow information. However, there is a trade-off between complexity and utility where, after some threshold is reached, adding complexity decreases utility.

One potential limitation in our study is that we provided participants with all four representations simultaneously. This presentation can introduce a potential confound by obscuring the source of information used when creating change effect descriptions. While participants were asked to identify which representations they used, they may have extracted some information from each representation, perhaps subconsciously. Thus, although we intended to collect an objective measure of utility free of participant's perceptions, the measure of applied usefulness identifies where participants perceive they are obtaining needed information and may not be entirely accurate.

As well, we are measuring comprehension with respect to two change effect descriptions and a program summary. It is possible that the representations did increase program understanding, but that the questions asked to determine comprehension did not require the use of the newly gained understanding. As in any task-oriented study on comprehension, we only measured comprehension with respect to the tasks and did not necessarily measure increases in overall comprehension. While the use of program summaries minimizes this possibility, there remains the likelihood that participants gained detailed low-level understanding that was not used to construct the summary.

## 5 Conclusion

It has been demonstrated that control-flow views of programs facilitate comprehension [10]. Additionally, it has been shown that, for visual programming languages, data-flow and control-flow representations lead to different types of program comprehension [6]. Data-flow leads to a more

abstract/functional description while control-flow leads to a more concrete/procedural description of the program. Given that Prograph is a data-flow VPL, the use of control-flow augmentation can bring the two areas of comprehension together and help programmers develop a more comprehensive program model.

From the experiment described here, it appears that nesting strikes a balance between increased complexity and the inclusion of additional control-flow information. While participants found that more detailed inter-method control-flow augmentation appeared to be useful, this amount of augmentation adds excessive complexity and hinders comprehension, at least in the short-term. For larger applications with considerably more methods, it is probable that the increased complexity of the fully augmented representation would be even greater.

A great deal of future work still needs to be done to confirm the results of this study. The sample size, while adequate, would benefit from being increased. As well, a clearer, more intuitive hierarchical representation should be investigated. The lack of clarity in the existing representation does not permit the hierarchical representation to be accurately compared to the nested representation.

When pilot testing the stimulus, participants were shown a third application. This application was approximately twice the size of the two that were included in the final experiment. While it would be beneficial to explore the effects of the representations with a longer, more realistic program, it was necessary to drop the third application for practical reasons. Without some form of renumeration, we were unable to recruit participants who were willing to complete the longer experiment. In future research, we intend to explore the scalability of the representations and their utility for longer and more complex programs.

Our results suggest that there is a trade-off between complexity and utility. The nested representation, which clearly displays intra-method calling sequences, satisfies this trade-off by adding control-flow information to a visual data-flow language. While nested representations are not highly effective for program development environments, we believe that they show significant promise for aiding comprehension through the merger of control-flow and data-flow information.

When programmers need access to data-flow information, the commercial success of visual data-flow languages suggests an effective data-flow representation can be based on one of these languages. Our research indicates that Prograph/CPX, a visual data-flow language, provides the best support for program comprehension when augmented with inter-method control-flow information. Additional augmentation, such as intra-method control-flow, adds excessive complexity and hinders comprehension. We examined two methods for expressing inter-method control-flow and found nesting to be an intuitively understood technique for effectively representing method calling sequences and positively influencing program comprehension.

## References


[1] K. Anjaneyulu and J. Anderson. The advantages of data-flow diagrams for beginning programming. In C. Frasson, G. Gauthier, and G. McCalla, editors, *Second International Conference on Intelligent Tutoring Systems*, pages 585–592, 1992.

[2] C. Corritore and S. Wiedenbeck. What do novices learn during program comprehension? *International Journal of Human-Computer Interaction*, 3:199–222, 1991.

[3] L. Freeman. A refresher in data flow diagramming: An effective aid for analysts. *Communications of the ACM*, 46(9):147–151, 2003.

[4] D. Gilmore and T. Green. Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, 21:31–48, 1984.

[5] J. Good. The 'right' tool for the task: An investigation of external represention, program abstractions and task requirements. In W. Gray and D. Boehm-Davis, editors, *Empirical Studies of Programmers: Sixth Workshop*, pages 77–98, Norwood, NJ, 1996. Ablex.

[6] J. Good. VPLs and novice program comprehension: How do different languages compare? In *Symposium on Visual Languages*, pages 262–299. IEEE, 1999.

[7] T. Green, M. Petre, and R. Bellamy. Comprehensibility of visual and textual programs: A test of of superlativism against the 'match-mismatch' conjecture. In J. Koenemann-Belliveau, T. Moher, and S. Robertson, editors, *Empirical Studies of Programmers: Fourth Workshop*, pages 121–146, Norwood, NJ, 1991. Ablex.

[8] K. Lloyd and D. Jankowski. A cognitive information processing and information theory approach to diagram clarity: A synthesis and experimental investigation. *The Journal of Systems and Software*, 45:203–214, 1999.

[9] I. Millet. Technical note – a proposal to simplify data flow diagrams. *IBM Systems Journal*, 38(1):118–121, 1999.

[10] R. Mosemann and S. Wiedenbeck. Navigation and comprehension of programs by novice programmers. In *International Workshop on Program Comprehension*, Toronto, ON, 2001. IEEE.

[11] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[12] S. Steinman and K. Carver. *Visual Programming with Prograph CPX.* Manning Publications Co., 1995.