

CSC 2514 : Human-Computer Interaction

A Survey of Visual Languages in Natural Programming

Yaroslav Riabinin (y.riabinin@utoronto.ca)

Tuesday, December 4, 2007

This paper is a survey of current research on visual languages in the context of natural programming. The goal of such research is to improve the interaction of users with programming languages by creating technologies that make this exchange more “natural” – closer to the way nonprogrammers expect the software to work.

In order to provide a broad overview of this research, typical programming tasks are identified. These include writing a new program and making changes to an existing program. What follows is a discussion of the problems that arise when these tasks are carried out using the methods currently available, such as the difficulties of learning the syntax of a new programming language. Then, a solution to these problems is presented for each task. The common theme in all of the solutions is that they somehow facilitate the interaction between the human and the computer by making it more “natural” with the aid of visual expressions. Each of these solutions is accompanied by a brief summary of the evaluation of the research. Finally, prospects for future work on this topic are discussed.

What is Natural Programming?

Originally, the main concern when designing a programming language was functionality. Designers never really considered usability issues – such as whether the language is easy to learn or whether it allows users to identify errors in the code efficiently. Thus, users of modern programming languages face all sorts of problems when attempting to carry out basic tasks, such as writing code for a simple sorting algorithm.

According to Myers [7], the problem is that despite advances in human-computer interaction research, not much has changed in the field of programming language design. The same mechanisms are currently used for looping, conditionals, assignments, and debugging as have been used for the past 60 years.

Given the circumstances, it may be time to consider how nonprogrammers envision their interaction with software and then use this insight to design more user-friendly programming languages. This endeavour is called natural programming and could be defined as follows: “the Natural Programming approach is an application of the standard user-centered design process to the specific domain of programming languages and environments” [7].

What is a Visual Language?

Programs are generally defined using textual elements – by writing lines of code, which are instructions to the computer. However, this may not be the most efficient way to create software. An alternative method is to specify programs by manipulating graphical elements. Therefore, a visual language is a programming language that requires the user to arrange a set of visual expressions in virtual space in order to specify the actions that are to be performed by the computer.

Previous work

Myers [7] offers an introduction to natural programming languages and environments, discussing the motivation behind the endeavour and presenting two examples of technologies that adopt this approach to programming language design. However, there are no papers that examine the issue of natural programming on a broader scale – looking at various programming tasks – and especially with reference to visual languages as possible solutions to current usability problems in human-programming language interaction.

Present work

In this paper, I give an overview of natural programming – and in particular, visual languages – by considering four typical programming tasks, outlining problems that arise when users carry out these tasks using the current available methods, and then presenting a more “natural” solution to these problems. I also provide an evaluation of each solution and discuss future work that could be done on visual languages and natural programming.

The following is a discussion of several typical tasks that are carried out by a user of a programming language. Problems with each task are identified and a solution is proposed, along with an evaluation of the solution.

Programming Tasks

1: *Writing a new program*

Problems

Creating a new program is arguably the most basic task that a programmer can perform. However, even beginning to write code can be highly problematic, given the current programming languages. The problem that novice programmers encounter is learning the complicated syntax of a given language. This usually consists of memorizing conventions and rules regarding how to combine strings to generate meaningful code – for example, where to put brackets or semi-colons.

Moreover, even a slight error in the code – a missing comma, for example – could cause syntax errors, which means that the program will not work properly. This often causes the programmer to become frustrated. As a result, many students who begin to learn about programming eventually give up and lose interest. That is why it has been found that incoming Computer Science majors in U.S. colleges declined more than 60% between 2000 and 2004, as reported in a promotional video for “Alice” software (www.alice.org).

Solution

Researchers at Carnegie Mellon University (CMU) have created a 3D programming environment that engages novice programmers by reducing the complexity of details that must be learned [2] [3]. This educational software is called “Alice” and it is an example of a commercially successful visual language.

It works by having users add 3D objects to a virtual world and write programs to generate animations. It is essentially an object-oriented programming language, like Java. However, it has an interactive interface that displays an object tree of the objects in the current world, the initial “scene”, a list of events in the world, and a code editor. Users write programs by using the mouse to drag and drop graphical tiles into the editor. These visual expressions correspond to instructions in a standard programming language – such as assignment statements, loops,

conditionals, and so on. Users can run their animation immediately to see the results of their work.

Therefore, “Alice” has two major benefits:

(1) it prevents the user from making syntax errors by restricting the coding process to dragging and dropping syntactically correct visual expressions;

(2) it provides constant feedback through 3D animations, which allows the user to quickly identify errors and fix them.

The result is that the initial frustration of learning a new programming language is greatly reduced.

Evaluation

“Alice” software has gained commercial popularity and it is used as a teaching aid in many U.S. schools. However, empirical user studies have also been conducted in order to evaluate the effectiveness of this product.

In Cooper [3], the following experiment was conducted: in the 2001-2002 school year, 21 of the weakest CS majors at Ithaca College and Saint Joseph’s University were invited to take a course that introduces programming concepts using “Alice” software. This course was taken prior to, or concurrently with, CS1, which is a basic computer science class.

11 of the 21 students took the course and 10 did not.

The results indicate that the students who took the “Alice”-based course had a much higher grade in CS1 than those who did not take the course. Moreover, of the 11 students who took the course, 91% proceeded to take the next computer science class, CS2. Of the 10 students who did not take the course, only 10% continued on to CS2.

2: *Debugging a program*

Problems

Although this task continues to be one of the most common and costly programming activities, little has been done over the past 30 years to improve debugging techniques [4]. Programmers continue to manually introduce print statements to display program output at different stages and use breakpoints and code-stepping to figure out why their program does not function properly.

It has also been found that many errors come from false hypotheses that users have about existing errors in their code [4]. In other words, since users are unable to properly identify the problem, they can jump to false conclusions about why their program is not working, which leads to even more errors when users try to fix a problem that does not exist.

Solution

In response to these concerns, researchers at CMU have devised a novel paradigm called “Interrogative Debugging” (ID) [4]. It directly addresses the issue of hypothesizing about program errors by allowing the user to ask “why did” and “why didn’t” questions about their program’s runtime failures. Thus, any implicit assumptions about what did or did not happen are made explicit.

In order to test this paradigm, the authors developed Whyline – a **W**orkspace that **H**elps **Y**ou **L**ink **I**nstructions to **N**umbers and **E**vents. It is a system for debugging programs written in “Alice”, which was introduced in the previous section. Once a world is run, a Question Menu appears where the user can make queries regarding runtime events. The user can select “why did” or “why didn’t” questions from a hierarchical drop-down menu, such that code related to the question is highlighted.

Once a question is asked, the system analyzes the runtime actions that did or did not happen and provides an answer to the question by displaying a graph of the actions that are relevant to the problem, while excluding irrelevant actions. Therefore, the user simply needs to examine the execution history that is presented in order to locate the error.

Evaluation

A user study that was conducted in Ko [4] evaluated the overall usefulness of Whyline. To investigate this question, 9 participants were asked to make a Pac-Man game with 6 distinct specifications. Whyline was made available to 5 of the participants, while the other 4 had to use regular debugging strategies.

The results show that Whyline significantly decreased debugging time, by an average factor of 7.8. Also, participants that used Whyline were able to complete more tasks within the allotted time period. More specifically, there was a 40% increase in the tasks completed.

Additional Research

It is worth mentioning here that research on Interrogative Debugging was continued at CMU in the form of Crystal – **Clarifications Regarding Your Software** using a **Toolkit, Architecture and Language** [7]. The authors of Whyline extended their framework to other domains, using Microsoft Word as an example of a modern application that has many automatic features that often remain hidden and mysterious to the user. The purpose of Crystal, then, is to make all these features “crystal clear” by allowing the user to ask “why did” and “why didn’t” questions.

3: *Making changes to a program*

Problems

Once a program is already written, the programmer often needs to make certain modifications to the code. For example, one might need to fix a bug that has been detected, or to change the functionality based on new design requirements. However, even a conceptually simple change could require numerous edits and could have far-reaching effects [1]. This makes the task tedious and creates the risk of introducing errors into the code. Although some tools have been designed to address this problem and to automate the transformation process, most of these have not been very successful, due to their difficulty of use.

Solution

In response to this problem, researchers at the University of California at Berkeley have developed a novel program manipulation paradigm that makes use of visual expressions to support a broad range of code-changing tasks, which are now carried out in an intuitive manner [1]. The visual language that was created is called “iXj”. It represents patterns visually by surrounding code fragments (in Java) with labeled graphical boxes. Each box marks a structural entity – such as an object, a method, or a set of arguments. Inside the boxes, patterns can contain not only code fragments, but wildcards that match a broader set of structural elements (for example, all objects whose type is a subtype of `java.util.Vector`). Below the visual representation of the pattern is a transforming action that is specified by the user and that is presented in a shaded input field.

The “iXj” prototype was implemented inside Eclipse. The main addition was the Transformation Editor, which allows the user to manage the transformation patterns and the actions to perform. Also, the standard Eclipse views were augmented with pattern matching and transformation feedback, highlighting matches in the source code.

Evaluation

In order to evaluate the usefulness of the visual transformation language, Boshernitsan [1] conducted a usability study with 5 Java programmers. Participants were first trained to use the transformation system. Then, they were asked to complete a short code editing task and to fill out an evaluation questionnaire.

The results of the study indicate that the participants' performance increased with experience with the tool. According to the results of the questionnaire, programmers found the interface to be intuitive, easy to learn and effective on the code editing task.

4: *Visualizing ideas as program code*

Problems

Although it would be very useful to be able to transform a mental plan in familiar terms (natural language, for example) into one that is compatible with the computer, this task is a difficult one. Moreover, this issue is related to the concept of *direct manipulation*, which is a key principle in making user interfaces easier to use [7]. The problem is that there is currently no simple way to express ideas that are in the user's language directly as programming code. There must always be an intermediate step of converting descriptions of what the program should do into executable code.

Solution

In response to this problem, researchers at MIT have developed a visualization tool called "Metafor" [5], which automatically performs a conversion between a story that the user writes in natural language and program code. This code, however, is not actually executable. Rather, the system generates *scaffolding code* that is underspecified, but succeeds at providing the user with immediate feedback about what their code *roughly* looks like. As the user enters a story into "Metafor", the system continuously generates a "visualization" of the text in the form of an outline of the code (a skeleton).

Currently, the "Metafor" interface consists of four windows: (1) where the narrative is being entered; (2) interaction log that records the full story; (3) a representation of the parse tree, for debugging purposes; and (4) the generated code, in Python.

According to the authors, "Metafor" achieves the following two goals:

- (1) it helps novice programmers get started on their program;

- (2) it helps intermediate programmers with brainstorming, before they attempt to write the actual code.

Evaluation

In a 13-person study performed in Liu [5], both nonprogrammers and intermediate programmers were asked to estimate the time it would take them to program a simple Pac-Man game. Then, they were asked to write a short story describing the game using “Metafor” and then re-estimate the task completion time to see if there was any improvement. Participants were also asked how likely they were to brainstorm with “Metafor” versus paper.

The results show that nonprogrammers estimated a 22% reduction in task time when using “Metafor”, while intermediate programmers estimated it to be 11%. Also, nonprogrammers rated themselves as “likely” to “very likely” to use “Metafor” for brainstorming. Intermediate programmers were more confident in their paper brainstorming skills, but they still indicated that they were 31% more likely to use “Metafor”.

This paper represents a first attempt to provide a broad overview of the topic of natural programming and visual languages. The typical programming tasks that have been identified are by no means exhaustive and there may be other tasks that are unreasonably difficult to carry out using current programming languages and environments.

Hopefully the problems raised in this work illustrate the need for a more “natural” approach to programming language design, exemplified by the use of visual expressions to specify program instructions. Of course, there may be other ways to achieve the goals of natural programming and it is important to explore these alternatives in future research.

The following is a list of questions that could form the basis for future work on visual languages and natural programming:

- Can we create a visual programming language, such as “Alice”, that would rival standard programming languages (e.g., Java) in its functionality? Could such a language replace the currently dominant programming paradigms?
- Can we create a system that converts stories written in natural language into executable code (as opposed to just “scaffolding” code)?
- Can we create a debugging interface where the user is not restricted in the questions that could be asked? Instead of selecting “why” or “why not” questions from a menu, could a user be able to type in a question of their choice and get a response from the program?

1. Boshernitsan, M., Graham, S. L. and Hearst, M. A. (2007). **Aligning Development Tools with the Way Programmers Think About Code Changes.** *ACM Conference on Human Factors in Computing Systems*, San Jose, California, USA, April 2007.
2. Cooper, S., Dann, W., and Pausch, R. (2000). **Alice: a 3-D Tool for Introductory Programming Concepts.** *CCSC Northeastern Conference on the Journal of Computing in Small Colleges*, Ramapo College of New Jersey, Mahwah, New Jersey, USA, 107-116.
3. Cooper, S., Dann, W., and Pausch, R. (2003). **Teaching Objects-first in Introductory Computer Science.** *SIGCSE Technical Symposium on Computer Science Education*, Reno, Nevada, USA, February 19-23, 191-195.
4. Ko, A. J. and Myers, B. A. (2004). **Designing the Whyline: A Debugging Interface for Asking Questions about Program Failures.** *ACM Conference on Human Factors in Computing Systems*, Vienna, Austria, April 24-29, 151-158.
5. Liu, H. and Lieberman, H. (2005). **Metafor: Visualizing Stories as Code.** *ACM International Conference on Intelligent User Interfaces*, San Diego, CA, USA, January 9-12, 305-307.
6. Myers, B. A., Pane, J. F. and Ko, A. (2004). **Natural Programming Languages and Environments.** *Communications of the ACM*, special issue on End-User Development, September, Vol. 47, No. 9, 47-52.
7. Myers, B. A., Weitzman, D., Ko, A. J., Chau, D. H. (2006). **Answering Why and Why Not Questions in User Interfaces.** *ACM Conference on Human Factors in Computing Systems*, Montreal, Canada, April 24-27, 397-406.