Multiple views of control flows as a tool to improve programmer's understanding of interactive software

Fabien André

Multiple views of control flows as a tool to improve programmer's understanding of interactive software

Fabien André

Université de Toulouse - ENAC - IRIT
7 avenue Edouard Belin, 31400 Toulouse, France
fabien.andre@enac.fr

Abstract. My work addresses the question of how to improve programming notations and environments used for developing interactive systems. I focus on representing the control flow in interactive programs in the hope that it will improve the programmer's ability to understand how their programs work. My thesis is that offering programmers the ability to create and manipulate multiple ad-hoc views of the control flow is the right way to go, where current programming languages and environment each choose one particular view.

1 Introduction

We are interested in GUI programmers, who build interactive software with high demands on behavior [19] and graphics [6]. One of their main difficulties is with external control flows: the execution order of instructions depends on users' actions or other external conditions. This conflicts with the imperative functional paradigm, and one needs workarounds such as callbacks: subprograms do not call each other but ask to be activated under certain conditions. As a result, one can not follow the execution flow straightly, and need to track all subscriptions (problem known as the "spaghetti of callbacks"[21]).

This problem is particularly crucial when it comes to changing existing code. One needs to understand causal relationship before selecting the instructions that need editing: *"What should I change in code to change this background color?"*, *"What happens if I alter this line of code?"*.

In the next part, we will introduce a work scenario highlighting this problem. We will then review how others have addressed this problem, then present our angle and current results.

1.1 Work Scenario

Annette is coding an half-pie menu to navigate hierarchical data structure on the side of a tabletop system (as described in [13]). A colleague of hers has successfully implemented the menu with arrow button navigation. She is now adding a direct manipulation scrolling to her Menu. She notes a weird flickering in the motion of menu items.

The drag callback of the dragged MenuItem updates the MenuItem position, as well as the offset of the MenuLevel, other MenuItems are notified through an Observer **(problem: spaghetti of callbacks)**. The problem comes from Menu Item embedding a "slow in /slow out" animation in position accessor, *viz.* setting the position of an Item launches an animation to reach new position **(problem: machinery in seemingly innocuous code)**. This code was introduced for the step-by-step scrolling with buttons. The control flow interesting Annette is spread across two files, five functions, with inversions of control that make the control flow hard to follow.

2 State of the art

Studying programmer practice shows that they spend most of their time editing existing code, and thus trying to understand it [26]. A number of works have tackled program understanding in different communities: Programming Languages, Psychology of Programming, Software Engineering, Software Visualization.

A first class of tools promote a top-down approach to reduce *information complexity*[1]. This started with structured programming (that aimed at "intellectual manageability" [27]), and continued with Object-Oriented Programming and associated languages (Smalltalk, etc.) This approach is also at work in scene graph-based GUI toolkits. With this structured approach a limited set of object is manipulated at each level of structure in the program.

A second class of tool claim for a paradigm shift to empower the user by gathering instructions according to higher level relationship (*increase locality*). This class includes logical programming (Prolog, constraints in GUI Toolkits [20]), dataflow (Intuikit[6], ICon [9]), or reactive programming (Squeak[4], Esterel [10]). In the same way, some use existing models such as Statecharts [12, 2] or Petri nets [22] to describe GUI behavior.

Alternatively some take into account that programs are composed of a set of concerns that can not all be reflected in the main decomposition. They try to *increase locality* of so-called cross-cutting concerns or control flows. This multidimensional structure may be achieved by a conceptual paradigm shift (Aspect-Oriented Programming[14], Role Modeling[1]), other focus on presentation (CodeBubble[3], CodeCanvas[8]) or static analysis of code (Concern Graph[25]). Ko worked on debugging tools and highlight the potential of providing causal relationship abstraction to help developers understand a program [16].

Ultimately, studies shows programmer organizing his work in task and sub-task and determining a working set of artifact concerned by his current task [15].

3 Chosen Approach

3.1 Goal

My review of the state of the art suggests an evolution toward understanding the concept of locality as relative to a given task. At one precise moment, programmers are only interested in a few causality relations: *"how moving a finger will scroll this lists elements?"*, *"which components alter this variable value?"*, etc. This can be illustrated by the two scenarios below:

Version1: Annette ask the system to show link between finger coordinate and a circle coordinate. She notes that the path include a "smooth animation" dataflow boxes used to provide a "slow in / slow out" animation for the step by step version. This combined with sampling from tactile screen lead to flicker.

Version2: In a dataflow environment, Annette plugs the coordinates of fingers and circle on to a plotting view, then performs a drag and inspect the two curves. At a sufficient zoom level, she sees that sampling produce a common piecewise-constant function and the other curve is smoother following a "rotated atan" pattern. She identified this pattern as "slow-in / slow-out" and starts locating it to correct the bug.

In this context, my approach consists in trying to design programming tools that support relative locality, using design techniques from the user interface design community. Using scenarios such as the one above, I try to formulate the problem using a set of abstractions that capture the requirements, then I apply iterative design and evaluation methods.

3.2 Thesis: all we need is view

We studied other framework or guidelines for programming tools, such as Cognitive Dimensions of Notation [11], Cognitive Question in Software Visualization [23] or guidelines for error-preventing programming system [17]. We compared usability requirement of computing-oriented and interactive-oriented programming tools in [18] to illustrate similarities and differences in requirement in these two families of systems. We concluded on the importance of locality that we define: *"A local representation of a relationship is a representation in which a visual variable efficiently support perception of this relationship."* Our definition of view is a variant of Reenskaug's

[1] italicized words are concepts from the framework sketched in [18]

definition in MVC[24]: *"a view is a (visual) representation of its program. It would ordinarily highlight certain attributes of the program and suppress others. It is thus acting as a presentation filter"*. Representation may include dynamic data extracted from the program lifecycle: history, translations, execution machinery[2] ; still in the idea of improving understanding of the program. Depending on the nature of data represented (static or dynamic) or representation (invertible [7]), views may be editable and change underlying program.

Facing the variety of abstraction levels and concerns, my thesis is that in order to maintain intellectual manageability we need multiple views, one for each type of question that the programmer is addressing at a given time. The views can be made of text or graphics.

Thus we face the problem of providing representation and offering manipulation of the representation and of the program. We aim to provide a coherent set of views and handle to fit the programmer need to manage all the angles of his programs.

3.3 Method

I practice participatory design to target usable tools. I wrote down scenarios (as presented in 1.1) from interviews and observation of UI programmer. A part of these scenarios anticipate the arising need for dynamically configured multi-devices app (spreading for a while as proof-of-concept). Part of the conceptual complexity induced by such systems is addressed by a toolkit my lab is designing: an interacting component model assembled in a scene graph embedding behavior [5]. We used scenarios to imagine and design program visualization (textual or graphical, alterable or passive); editor and translator features, or more conceptual ideas.

4 Current results

Based on users' wishes, I first explored traditional text views — syntax of programming languages — with direct mapping to a scene graph. I explored different (not exclusive) tracks: emphasis on terseness through tiny syntax and default arguments, binding using visual clues (indentation), dataflow constraints.

```
r : Rectangle {
    x:10  y:10  w:20  h:20
    behavior: Statemachine {
        idle:State{
            Transition {on: r.press ; to: dragging }
         }
        dragging: State{
            Connector{ in: system.mouse.x out: r.x}
            Connector{ in: system.mouse.y out: r.y}
            Transition {on: r.release ; to: "idle"}
        }
    }
}
```

Fig. 1. Example of lightweight tree syntax. This program display a draggable rectangle.

I then worked on more pictorial views of scene graphs and developed a component tree view with an overlay representing dynamic transfer control across the tree. A concern with this tree-oriented views is the differentiation between hierarchy and control transfer relations. Hierarchy provide structure (abstraction and names) as well as control transfer between parent and descendent. Thus, when focusing on control transfer, the tree representation emphasizes a distinction where it is not needed.

[2] the running application is also a view of the program.

Consequently I investigated graph representations of components to avoid this difference. This work is a straightforward application of the above mentioned concept of view. The whole program is presented as a big and dense graph. This graph can be filtered different way: abstraction of nodes in higher-level nodes, selection of specific node of interest, selection of transfer of control (edge) according to their nature (initialization, dataflow, network)[3]. Figure 2 shows an example of extracting dataflow control transfers from a control flow graph. Based on this, I am exploring the hypothesis that language syntaxes each privilege some types of arcs and hide the others.

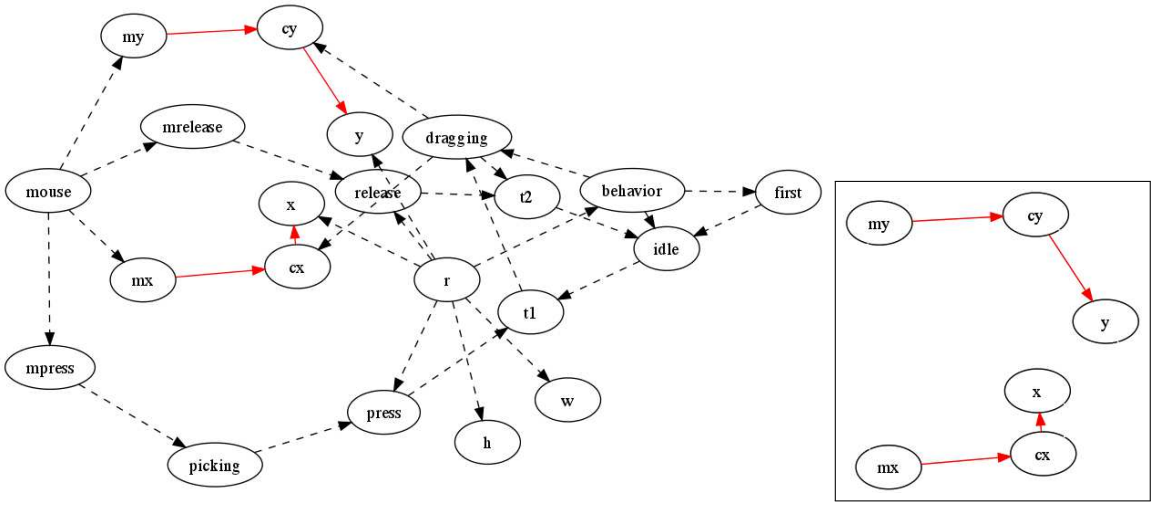

Fig. 2. Highlight of dataflow control transfer among the complete graph of interaction between entities.

5 Future work

We consider designing and implementing more complex pattern and transformation on the graph view. We are thinking of displaying profiling informations, as well as linking our view with existing models representation (Statecharts, ICon dataflow[9]) either by dedicated views or through "morphing" on the graph. We are rather confident in the use of several notation to answer the varied task of programming.

Until now, I have evaluated my work informally using framework listed in 3. These frameworks relying on trade-offs between dimensions are useful at design-time but I would like to provide objective evaluation and comparison between my solutions and toward other tools. Thus I am seeking for qualitative (Bertin?) or quantitative criteria (LOC, what else ?) and wonder if experimental evaluation would be possible / profitable. Especially with the ultimate goal of comparing "whole programming systems".

6 Acknowledgements

I would like to thank my advisor Stéphane Chatty, as well as Stéphane Conversy and Christophe Hurter for their invaluable help, advice and encouragement writing this paper.

[3] thus, the hierarchy tree is a particular case of this graph: a filtered view of the whole program showing only hierarchical relationships.

References

1. Egil P. Andersen and Trygve Reenskaug. System design by composing structures of interacting objects. In Ole Lehrmann Madsen, editor, *ECOOP '92 European Conference on Object-Oriented Programming*, volume 615, pages 133–152. Springer-Verlag, Berlin/Heidelberg, 1992.
2. Caroline Appert and Michel Beaudouin-Lafon. SMCanvas: augmenter la boîte à outils Java Swing pour prototyper des techniques d'interaction avancées. pages 99–106, Montreal, Canada, 2006. ACM.
3. Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. Code bubbles. In *Proc. of ACM/IEEE - ICSE '10*, page 455, Cape Town, South Africa, 2010.
4. Luca Cardelli and Rob Pike. Squeak: a language for communicating with mice. *SIGGRAPH Comput. Graph.*, 19(3):199–204, 1985.
5. Stéphane Chatty. Supporting multidisciplinary software composition for interactive applications. In *Software Composition*, pages 173–189. 2008.
6. Stéphane Chatty, Stéphane Sire, Jean-Luc Vinot, Patrick Lecoanet, Alexandre Lemort, and Christophe Mertz. Revisiting visual interface programming: creating GUI tools for designers and programmers. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, UIST '04, page 267–276, Santa Fe, NM, USA, 2004. ACM. ACM ID: 1029678.
7. Stéphane Conversy. Improving usability of interactive graphics specification and implementation with picking views and inverse transformations. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011, IEEE Symposium on*. IEEE, 2011.
8. Robert DeLine and Kael Rowan. Code canvas. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, page 207, Cape Town, South Africa, 2010.
9. Pierre Dragicevic and Jean-Daniel Fekete. Support for input adaptability in the ICON toolkit. In *Proceedings of the 6th international conference on Multimodal interfaces*, ICMI '04, page 212–219, State College, PA, USA, 2004. ACM. ACM ID: 1027969.
10. Jean-Daniel Fekete, Martin Richard, and Pierre Dragicevic. Specification and Verification of Interactors: A Tour of Esterel. In *Proc. of (FAHCI'98)*, Sheffield, UK, September 1998.
11. T. R. G Green. Cognitive dimensions of notations. In *Proceedings of the fifth conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and computers V*, page 443–460, Univ. of Nottingham, 1989. Cambridge University Press. ACM ID: 93015.
12. D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498. Springer-Verlag New York, Inc., 1985.
13. Tobias Hesselmann, Stefan Flöring, and Marwin Schmitt. Stacked Half-Pie menus: navigating nested menus on interactive tabletops. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, ITS '09, page 173–180, Banff, Alberta, Canada, 2009. ACM. ACM ID: 1731936.
14. G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es), December 1996. ACM ID: 242420.
15. Andrew J Ko, Htet Aung, and Brad A Myers. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, page 126–135, St. Louis, MO, USA, 2005. ACM.
16. Andrew J Ko and Brad A Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '04, page 151–158, Vienna, Austria, 2004. ACM. ACM ID: 985712.
17. Andrew J. Ko and Brad A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1-2):41–84, 2005.
18. Catherine Letondal, Stephane Chatty, Greg Phillips, Fabien André, and Stephane Conversy. Usability requirements for interaction-oriented development tools. In Joey Lawrance and Rachel Bellamy, editors, *Proceedings of the 22nd Annual Workshop of the Psychology of Programming Interest Group PPIG 2010, Sep 2010*, pages 12–26. Maria Paloma Díaz Pérez and Mary Beth Rosson, 2010.
19. B. Myers, S. Y Park, Y. Nakano, G. Mueller, and A. Ko. How designers design and program interactive behaviors. In *Proc. of VL/HCC 2008*, pages 177–184. IEEE, September 2008.
20. Brad A. Myers. A new model for handling input. *ACM Trans. Inf. Syst.*, 8(3):289–320, 1990.
21. Brad A. Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. pages 211–220, Hilton Head, South Carolina, United States, 1991. ACM.
22. Philippe Palanque. Petri net based design of User-Driven interfaces using the interactive cooperative objects formalism. *DSVIS*, 1994.
23. M. Petre, A. F Blackwell, and T. R. G Green. Cognitive questions in software visualisation. 1996.
24. Trygve Reenskaug. Models - views - controllers. Technical report, XEROX PARC, December 1979.
25. Martin P Robillard and Gail C Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proc. of ICSE '02*, page 406–416, Orlando, Florida, 2002. ACM.
26. Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '97, page 21–, Toronto, Ontario, Canada, 1997. IBM Press.
27. Niklaus Wirth. On the composition of Well-Structured programs. *ACM Comput. Surv.*, 6(4):247–259, December 1974. ACM ID: 356639.