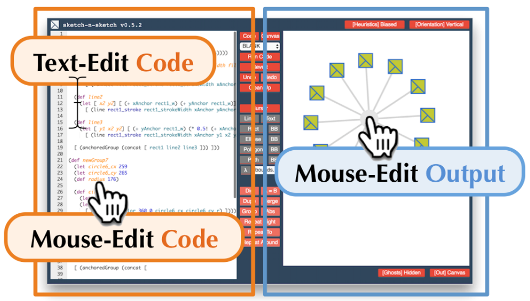
Ravi Chugh | Research Statement

Direct Manipulation Programming Systems



The typical workflow in existing programming languages involves an iterative process of text-editing, compiling and executing, viewing the output, and returning to the text buffer to make edits in order to effect desired changes to the program output. This disconnected workflow (a) limits the creativity and pace of expert programmers and (b) shuts out millions more users that, though not programmers, nonetheless create a variety of complex digital objects using direct manipulation, graphical user interface (GUI) based software. The primary goal of my research is to develop new interactive capabilities for writing programs, to bridge the gap between programming languages and direct manipulation interfaces—what I call *direct manipulation programming systems* [1].

To achieve this, my research program comprises three lines of work at the boundaries of **programming languages**, **software engineering**, and **human-computer interaction**: (1) To design program synthesis algorithms that help construct high-level, readable programs through direct manipulation interaction with examples of intended output. (2) To design structured code editors that augment text editing with intuitive interfaces for performing a variety of complex program transformations. (3) To design and implement interactive programming environments for a variety of application domains, to demonstrate that expressive programming languages and interactive direct manipulation interfaces need not be mutually exclusive. Together, the goal is to lay the foundations for programming environments that make experts more productive and make programming more accessible to novices. To begin exploring these directions, my students and I have developed Sketch-n-Sketch, a direct manipulation programming system for creating Scalable Vector Graphics (SVG). Sketch-n-Sketch is depicted in the screenshot above and available on the web at `http://ravichugh.github.io/sketch-n-sketch/`. Sketch-n-Sketch comprises work along all three fronts outlined above.

1: Program Synthesis via Direct Manipulation of Output

In the typical iterative "Edit-Run-View" workflow, an Edit step is often part of a *Prototype* phase to explore and set up an initial solution, a *Repair* phase to fix any remaining issues, or a *Refactor* phase to clean up the code to make it more readable and reusable. Our goal is to support common code transformations in these phases by allowing the user to specify examples of the desired change (*i.e.* using the output pane of the programming environment). Compared to many *programming by example* systems, we would like to generate program fragments that are high-level, idiomatic, and readable code in the source language—retaining as much of the style and structure of existing code as possible—so that the user can continue to develop the program and its output in synchrony. In Sketch-n-Sketch, we have identified common Prototyping, Repair, and Refactoring tasks that arise when programming SVG, and we have designed direct manipulation tools that trigger corresponding program updates.

Prototyping = Drawing Shapes. The Prototyping phase consists of drawing shapes directly in the canvas. Sketch-n-Sketch automatically generates template starter code, designed with formatting and names that are easy to read and manipulate [6]. This allows users to begin the design process using what feels like a traditional drawing editor, but by creating an initial program that is set up for subsequent text- and direct-manipulation edits.

Repairing = Relating + Adjusting Design Parameters. There are several common Repair edits when programming SVG. After drawing initial shapes with separately defined attributes (such as positions, sizes, and colors), a common operation is to relate attributes among shapes, for example, using shared variables instead of hard-coded constants. To introduce such relationships in the program requires two parts: to identify where in the program the desired relationship can be defined, at a place which has a common scope for all the relevant values; and then to fill in the desired relationship. The first part is boilerplate, tedious, and error-prone, whereas the second is more difficult because of the vast number of relationships the user may intend. Sketch-n-Sketch provides a *Relate* direct manipulation tool that, given a set of selected attributes in the output, automates the first part, introducing new variables in the innermost scope with a *hole* for the relationship between their values [6]. Certain relationships, like equality and equi-distance, can be filled in automatically with domain-specific solvers and search-based term enumeration of library components. When Sketch-n-Sketch cannot infer the desired relationship, the user can manually text-edit the hole expression.

Another common Repair operation is to adjust attribute values, or design parameters. Sketch-n-Sketch provides a feature called *live synchronization*, which allows the user to directly change these attributes in the output (as in a normal drawing editor) while appropriate constants in the program are changed to match [2]. The program is re-run during the interaction, so that the user can see the overall changes immediately. When there are ambiguities about what constants to change—for example, if a position `101` in the output is computed from a program expression `(let (base, offset) = (100, 1) in (+ base offset))`—Sketch-n-Sketch uses simple heuristics for automatically choosing to vary either the value of `base` or `offset`. Although simple heuristics do not always match the user's intent, the immediate feedback about the change helps make it easy for the user to try other alternatives in order to effect the desired change.

Refactoring = Grouping + Abstraction. Once the desired relationships have been encoded in the program, it is common to group, copy-paste, and adjust the shapes as needed. Programmatically, this corresponds to collecting all the relevant definitions into one that generates the list of expression, and then turning this single definition into a function abstracted over the design parameters. Sketch-n-Sketch provides transformations for these steps, using heuristics to decide which of the many potential constants to make design parameters [6]. Using these features, the user can group shapes in a similar way as in drawing editors, to create a function that can then be called with different arguments to create different variations. Together with live synchronization, these features provide a rapid way to "copy-paste-and-change" complex patterns generated programmatically.

Future Work. With this workflow, the user is able to turn the idea for the design into a high-level, reusable program with a large degree of automation from the system. We have written over 2,000 lines of code in Sketch-n-Sketch; many of these examples would have been tedious and extremely time-consuming to implement using existing programming languages or existing direct manipulation systems.

The program synthesis techniques in Sketch-n-Sketch represent only a first step towards our long-term vision for direct manipulation programming. Currently, the program update problem for each of these GUI operations is mostly syntactic, using heuristics to make choices in the face of ambiguity. While this works in some cases, there are many cases in which additional interaction from the user would be warranted. We plan to develop more powerful synthesis algorithms that search for likely desirable updates by ranking multiple factors—program size, as usual, but also factors such as output distances, how "different" program evaluation is, changes to dependency structure of the program, compatibility with subsequent likely transformations in the domain, and a history of interactions by the user and by a large population of users.

Despite using specific application domains—such as graphic design (SVG), web development (HTML5), and data manipulation and visualization—for motivation and evaluation, I believe that a general set of program update tasks will emerge that are useful across application domains, in addition to ones that are more specific to each. For example, the approach behind relating attributes, abstracting definitions, and adjusting parameters in Sketch-n-Sketch are largely independent of SVG-specific details. Furthermore, we are exploring synthesis algorithms based on nondeterministic evaluation and rewriting that perform domain-independent transformations.

2: Structured Editing via Direct Manipulation of Source Code

Whereas the first line of work is concerned with synthesizing program updates based on user interactions with the output of a program, this line of work is concerned with user interactions with the source code itself. Plain text continues to dominate as the universal format for programs in most languages. Although the simplicity and generality of text are extremely useful, the benefits come at some costs. For novice programmers, the unrestricted nature of text leaves room for syntax errors that make learning how to program more difficult. For expert programmers, many editing tasks fall within specific patterns that could be performed more easily and safely by automated tools.

Two bodies of work have, respectively, sought to address these limitations. *Structured editors* reduce the amount of unstructured text used to represent programs, with operations that create and manipulate structurally-valid programs, eliminating classes of errors altogether. Structured editing has not yet, however, become popular among expert programmers, in part, due to their cumbersome interfaces compared to plain text editors, as well as their restrictions that even transitory, evolving programs always be well-formed. An alternative approach witnessed by integrated development environments (IDEs) is to augment unrestricted plain text with support for *automated refactoring*. This approach provides experts both the full flexibility of text as well as mechanisms to perform common tasks more efficiently and with fewer errors than with manual, low-level text-edits. Although widely used in practice, automated refactoring tools typically do not help with the Prototyping and Repair phases, and they are underused due to usability challenges for identifying, invoking, and configuring transformations.

Text Editing + Direct Manipulation-Based Structured Transformations. We are developing a structure-aware code editor, called DEUCE, that is equipped with direct manipulation capabilities for automating structured program transformations [7]. The goal of our work is to enable a workflow that enjoys the benefits of text editing, structured editing, and automated refactoring. To achieve this, DEUCE augments a traditional text editor with (i) clickable selection widgets directly atop the program text that correspond to subexpressions and other relevant features of the program structure, and (ii) a lightweight, interactive display of potential transformations based on, and automatically positioned adjacent to, the current selections. With these direct manipulation features, the user can quickly and easily discover and invoke a variety of structured program transformations within a familiar, unrestricted text-editing workflow. The net effect is a programming workflow that is largely text-driven, but augmented with automated support for structured transformations (*e.g.* to introduce local variables, rearrange definitions, and introduce function abstractions) that are tedious and error-prone (*e.g.* because of typos, name collisions, and mismatched delimiters), allowing the user to spend keystrokes on more creative and difficult tasks that are harder to automate.

Our initial implementation of DEUCE has been developed in the context of SKETCH-N-SKETCH for several reasons. First, there is a relative lack of structured editing and refactoring tools for functional languages, compared to object-oriented ones (*e.g.* Eclipse for Java). Second, while the existing output-directed synthesis features in SKETCH-N-SKETCH attempt to generate program updates that are readable and which maintain stylistic choices in the existing code, the generated code often requires subsequent edits, *e.g.* to choose more meaningful names, to rearrange definitions, and to override choices made automatically by heuristics; DEUCE provides an intuitive and efficient interface for performing such tasks. Furthermore, the abilities to directly manipulate code and output are complementary, both contributing to the interactivity of the programming environment. Nevertheless, most of the features provided in DEUCE work for arbitrary programs (not just those used to compute SVGs) and, thus, may be adapted and extended to code editors for other programming languages.

Future Work. There are several opportunities to design additional direct manipulation interactions with the source text. While our current approach displays which transformations are applicable based on what items the user has already selected, it would be even better to anticipate what transformations the user may wish to perform even before all items are selected. The challenge here is to display suggestions for selections and transformations that may be of interest without cluttering the display and overwhelming the user with too many choices.

One benefit of our design is that it is easy to instantiate with different sets of transformations, which could enable domain-specific program transformations that are useful for particular tasks, projects, and teams. To make our approach more extensible, it would be useful to design a framework for defining rewrite rules and appropriate side conditions, so that DEUCE may be instantiated with different transformations in different settings without having to implement each inside the tool.

DEUCE provides a lightweight way—a quick keystroke—to toggle between text- and mouse-based editing. However, the complete separation between modes has limitations. As soon there are any character changes to the text, the structure information (and any active transformations) become invalidated. It would be better to incorporate an incremental parsing approach to tolerate some degree of differences in the source text, so that as much of the structure information as possible can persist even while the code has been changed with text-edits. This would help to further streamline, and augment, the support for structured editing within the full, unrestricted text-editing workflow.

3: Application Domains

Our work on synthesizing program updates based on direct manipulation of output and direct manipulation of code has been, so far, set in the context of SVG. But the tension between programming languages and direct manipulation crops up in every direction—web development (*e.g.* JavaScript vs. Dreamweaver), data manipulation (*e.g.* R vs. Excel), data visualization (*e.g.* D3 vs. Excel), word processing (*e.g.* LaTeX vs. Word), presentations (*e.g.* Slideshow vs. PowerPoint), 3D graphics (*e.g.* OpenGL vs. SketchUp), and more. Given the initial results obtained by the SKETCH-N-SKETCH project, I believe **direct manipulation systems for most application domains ought to be direct manipulating *programming* systems**. In addition to helping experts, I see two potential benefits for novices. The first is that the immediate, "two-way" connection between programs and output could be a useful pedagogical tool for motivating why and teaching how to program. Furthermore, although not every user will or should want to work directly with the program, using programs as the representation format can make it easier to blur the boundaries between what the user creates and what is built-in to the user interface [6] and can make it easier for expert library writers to customize the user interface with domain-specific tools [2]. As with the program synthesis direction, I expect to develop a set of domain-independent user interface techniques for interactive programming that will work broadly across domains, combined with domain-specific techniques for each.

Research for Practice. To ground this pursuit, I plan to continue developing prototype direct manipulation systems for several application domains. My hope is to lay foundations that enable engineers to develop industrial-strength direct manipulation programming systems. Thus, in addition to disseminating ideas in academic settings, we will continue to develop and release our software in the open, present in conferences such as Strange Loop that attract a mix of software engineers and researchers, and distribute tutorials and videos that are less technical than our research papers and seminars on our project webpage. If successful, a stretch goal is to incorporate our techniques in a language like Elm, which has been an active playground for functional language design for web programming.

General-Purpose Direct Manipulation Programming Environments. All of the application domains described so far involve digital objects with inherently visual representations, making it easy to see the need for direct manipulation interfaces. I also see similar opportunities for directly manipulating "general-purpose" programs; it may be desirable to manipulate intermediate traces and states (as in a debugger), and then have the system synthesize updates that are consistent with the changes and match the user's intent. Furthermore, the structured editing features in Deuce begin to expose GUI-based features for transforming program text. There may be even more intuitive direct manipulation actions—such as drawing tools to select portions of text—or to sketch out the desired structure and layout of code, as a way to constrain the synthesis problem for constructing the desired program. In these ways, I imagine that direct manipulation programming techniques will be useful even in domains that do not have inherently visual output.

Types and Program Analysis for Untyped Languages

Earlier in my career, in graduate school, I designed several type systems [5, 3] and other program analyses [4] for JavaScript, where the lack of types, flexible semantics, and prevalence of dynamically-loaded code combine to make static reasoning difficult. Although my current research has shifted away from these directions, one potential reason to return to them—in light of my recent focus on direct manipulation programming—is the prevalence of dynamic languages used in data science (*e.g.* Python and R), web application development (*e.g.* JavaScript), graphic design (*e.g.* p5.js, inspired by Processing), shell scripting and systems administration (*e.g.* Bash and Perl), and document creation (*e.g.* LaTeX). Each of these (untyped) languages is used to write programs that produce content with large amounts of visual output, and many of these languages include novice programmers among their users. Therefore, types and static analysis for these languages could help facilitate the kinds of program synthesis algorithms we seek to design and deploy for direct manipulation programming systems.

References

1 R. Chugh. Prodirect Manipulation: Bidirectional Programming for the Masses. In *International Conference on Software Engineering Companion (ICSE-C), Visions of 2025 and Beyond Track (V2025)*, 2016.

2 R. Chugh, B. Hempel, M. Spradlin, and J. Albers. Programmatic and Direct Manipulation, Together at Last. In *Conference on Programming Language Design and Implementation (PLDI)*, 2016.

3 R. Chugh, D. Herman, and R. Jhala. Dependent Types for JavaScript. In *Object-Oriented Programming Languages, Systems, and Applications (OOPSLA)*, 2012.

4 R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged Information Flow for JavaScript. In *Programming Language Design and Implementation (PLDI)*, 2009.

5 R. Chugh, P. M. Rondon, and R. Jhala. Nested Refinements: A Logic for Duck Typing. In *Principles of Programming Languages (POPL)*, 2012.

6 B. Hempel and R. Chugh. Semi-Automated SVG Programming via Direct Manipulation. In *Symposium on User Interface Software and Technology (UIST)*, 2016.

7 B. Hempel, G. Lu, and R. Chugh. Lightweight Structured Editing with Direct Manipulation, April 2017. Draft.