Runtime Visualization for Model-View-Update GUIs

Geoffrey Litt
MIT CSAIL
glitt@mit.edu

ABSTRACT

Visualizing the runtime behavior of programs can help programmers with targeted debugging and general understanding. For understanding complex programs, visualizations abstracted from the low-level code are most helpful, but this introduces new challenges: how does the programmer specify what to visualize, and how do they visualize complex data structures which aren't just primitive values?

In this work, I present an approach to visualizing the behavior of user interfaces built with the Model-View-Update pattern. I present a prototype runtime visualization system built on the Redux library and argue that, by exploiting the natural abstraction characteristics of this application architecture, we can create useful runtime visualizations with minimal programmer effort.

Author Keywords
Program visualization, program understanding, debugging

INTRODUCTION

Much recent work in program visualization [17, 4, 5, 11, 7] focuses on low-level details: showing the values of individual variables, connected to individual lines of source code. This works well for small programs, and for helping novices understand the basics of programming. But these visualizations don't address the needs of more experienced programmers working with larger programs. Gaining a general understanding of a large program requires zooming out from individual lines of code.

This leads to the idea of *abstract program visualization*: creating abstract, program-specific views of runtime state or static code, to help someone debug or understand the program. This idea has been explored in the context of teaching algorithms [1, 16] and understanding the behavior of multithreaded Java programs [13, 15]. But abstract visualizations create a new challenge [14]: how can we enable the programmer to create program-specific abstract visualizations with minimal effort? On the one hand, overly generic visualizations (as used in most low-level visualization systems) will often fail to capture the higher-level meaning of the specific program. On the other hand, if a visualization takes too much work to create, it won't be realistic for programmers to create the visualization in practice.

I think a promising strategy for approaching this problem is to create runtime visualization systems coupled to a particular domain-specific framework or DSL. Frameworks and DSLs occupy an intermediate place between general-purpose languages and specific programs. They often impose a particular mental model, code architecture style, and other constraints that usefully narrow the space of possible programs relative to a general-purpose language. On the other hand, there are still many different programs that can be built in one framework, so the effort of building a visualization system can be amortized over thousands of programs rather than concentrated on a single one.

To concretely test this strategy, in this work I propose a runtime program visualization system for user interfaces built with the Model-View-Update (MVU) architecture [3], also commonly known as the Elm Architecture [2]. MVU encourages the state of the interface to be centralized in a single data structure, derived by running a pure reducer function over a stream of events.

This architecture has many practical benefits for program understanding and developer experience (e.g., automatically achieving time-travel debugging), and I think it has useful characteristics for abstract program visualization as well. In particular, MVU naturally encourages programmers to define abstractions that represent the essence of their application's behavior: 1) a stream of semantically meaningful events, 2) a state object that represents all the core state of the UI. My hypothesis is that it is possible to visualize MVU interfaces with relatively little additional effort from the programmer, because the architecture has already required them to do much of the work of abstracting.

I've prototyped a runtime visualization system on top of the popular Redux [12] library, which implements MVU in Javascript. Within the limited scope of this project, I've focused on making a prototype specifically designed to visualize the state of the TodoMVC demo application. I've designed some visualizations tailored to the state of that application, and through my own usage I've begun to gain a preliminary understanding what kinds of visualizations might be useful to programmers navigating execution traces of MVU applications.

CHI'20, April 25–30, 2020, Honolulu, HI, USA

ISBN 978-1-4503-6708-0/20/04...$15.00
DOI: https://doi.org/10.1145/3313831.XXXXXXX

Much future work remains to fully flesh out this idea, including developing a crisper understanding of the needs of programmers, validating this system against those needs, and generalizing the system so that it actually works with many Redux applications instead of just one demo app.

RELATED WORK

Reiss [14] provides a useful taxonomy of execution visualizations, with pointers to prior research. Some particularly relevant dimensions for this work include abstract vs concrete, and effort required to create the visualization.

Many systems have explored visualizing execution state at the level of individual source lines, including Learnable Programming [17], Python Tutor [4], Omnicode [6], Theia [11], and Theseus [10].

Some systems have explored somewhat more abstract views. Projection Boxes [Lerner [9] provides a way of selectively showing parts of application state, and Seymour [7] provides a "macro" visualization to generally show the layout of execution flow, in addition to a "micro" visualization.

This work aims to provide a much more abstract view of the application's behavior than any of these other projects, by avoiding doing any visualization at the level of individual lines of code.

Other systems have explored this kind of abstract program visualization, entirely disconnected from the source code. For example, Balsa [1] and Tango [16] show animated views of algorithms operating, and Jive [13] and Jove [15] visualize various high-level projections of the execution of Java programs, e.g. when different threads are running.

I'm not aware of much prior research on abstract program visualization for user interfaces, although I still need to do a fuller literature review. UI performance analysis tools or debuggers like the Redux Dev Tools arguably fit into this category, but there aren't many tools that employ data visualization techniques to display the internal state of the application.

Hoffswell et al propose a system for visualizing runtime state inside Vega data visualizations [5]. That work fits into the category of visualizing state next to source code lines, but by integrating with a very high level domain-specific language, achieves more abstraction than visualization systems for general languages like Python. They also propose a design space for visualizations embedded in source code, which I plan to build on in this work.

VISUALIZATION DESIGN

Use cases

I had some prior experience with the Redux Dev Tools debugger, which provides the ability to inspect application state in Redux applications. From this personal experience, I identified two distinct use cases for a runtime visualization:

- **Localizing within a trace**: *Where do I need to rewind to, in order to inspect a particularly relevant point in an execution trace?* This is most often helpful when debugging a particular problem. Scrubbing back and forth while watching the UI change is often workable, but it's inefficient. Also, sometimes the relevant state isn't directly visible in the UI, so I need to dig into a JSON object at each point in time to understand whether I've found the right point in the trace.
- **Generally understanding program behavior over time**: *Overall, what happened as I interacted with the program?* Sometimes I'm not debugging a particular problem, and I'm more interested in just seeing general information about how a program is behaving over time. For example, this is helpful when explaining the system's behavior to a new programmer who's preparing to work on the system, or when I'm trying to learn the basics of a codebase myself.

These two goals partially overlap, but can also lead in different design directions. For example, localizing a specific point in a trace can benefit from a more active interrogatory approach (e.g. as explored in the Whyline system [8]), but general program understanding might benefit from a more passive style, more akin to reading documentation but accompanied by live demonstrations.

Data structures

Many concrete and low-level program visualizations focus on showing primitive values, especially numeric values. However, the state of an arbitrary MVU application often contains complex nested data structures, which contain many non-numeric values: booleans, strings, and enum values. One challenge for this system is to find ways to visualize these types of structures.

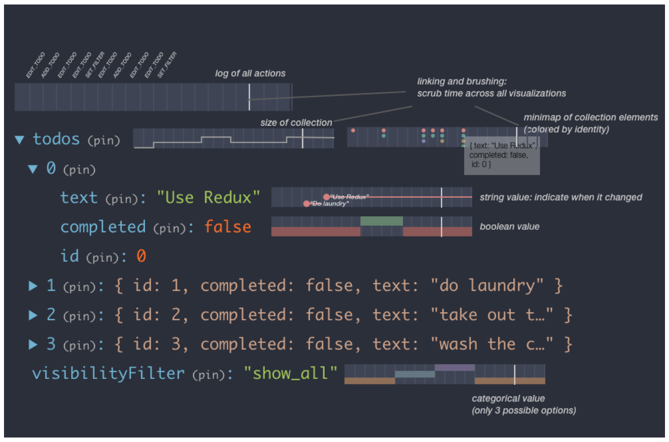
Context: TodoMVC

In order to focus my effort on concretely understanding the utility of visualizations, rather than building out infrastructure, I built a visualization system for a specific application: the TodoMVC GUI benchmark. TodoMVC is a basic todo list UI where the user can add, edit, delete, and complete todos. The user can also filter the list of todos shown to either active or completed ones.

The Redux implementation of TodoMVC stores an app state object which contains the list of todos, and the current state of the visibility filter. There are actions corresponding to each of the main user interactions listed above, e.g. "add todo" and "set visibility filter". Importantly, the Redux events capture an abstract, semantically meaningful picture of the user's interactions: when adding a new todo, the user's keystrokes are collected in the local state of a React component, and only a single "add todo" event is triggered in Redux once the user finally adds the new todo.

Overall layout

My initial idea, as shown in Figure 1, was to show the current state of the application as a nested JSON tree, and then to show small sparkline-style visualizations next to nodes of the tree. This design draws some inspiration from [5], but differs in that it uses visualization to annotate the application's state tree, rather than its source code. The advantage of this design is that it closely and directly links the current state to data from the execution history, but that link also causes thorny problems—for example, how do you deal with nodes that have disappeared


EDIT_TODO
ADD_TODO
EDIT_TODO
EDIT_TODO
SET_FILTER
EDIT_TODO
ADD_TODO
EDIT_TODO
EDIT_TODO
SET_FILTER
log of all actions
linking and brushing: scrub time across all visualizations
size of collection
minimap of collection elements (colored by identity)
{ text: "Use Redux", completed: false, id: 0 }
"Use Redux"
"Do laundry"
string value: indicate when it changed
boolean value
categorical value (only 3 possible options)

```
▼ todos (pin)
  ▼ 0 (pin)
      text (pin): "Use Redux"
      completed (pin): false
      id (pin): 0
  ▶ 1 (pin): { id: 1, completed: false, text: "do laundry" }
  ▶ 2 (pin): { id: 2, completed: false, text: "take out t…" }
  ▶ 3 (pin): { id: 3, completed: false, text: "wash the c…" }
  visibilityFilter (pin): "show_all"
```

Figure 1. JSON tree with inline sparklines

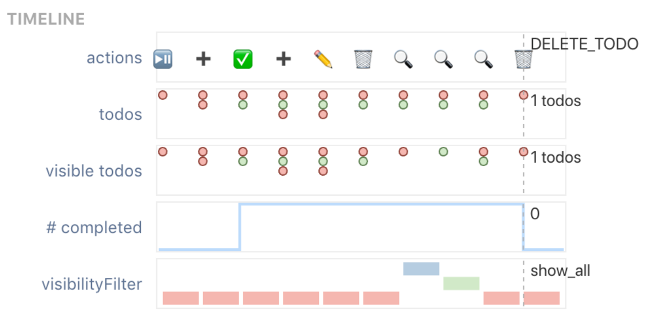


Figure 2. Timeline view of stacked visualizations

from the current state? Perhaps more concerningly, by tying the visualization to the *concrete* current state, it limits the ability of the programmer to create a customized abstract view, removed from the details of the state.

In my next iteration I switched to a different layout, shown in Figure 2: a vertically stacked list of small visualizations of state over time. Each visualization can display an arbitrary projection of the app's Redux state. Because the graphs are horizontally aligned, it's easy to see how different aspects of the app's state have changed in relation to each other. While I haven't implemented this yet, I imagine that programmers would be able to dynamically add visualizations to this list, specifying useful projections of app state, and deciding what type of visualization to use for each projection.

One thing lost in the timeline view is the concrete view of the app's entire state. It's still useful to see this, so I added a separate panel which displays that data. The user can scrub through history in the timeline, "pin" the app state at a particular point in time, and then use the separate state view to drill into the app's concrete state at that point.

Visualization Types

Here I describe the specific visualizations I prototyped for the timeline view. These are shown in Figure 2 from top to bottom. (The video demo linked on the project page might be an easier way to grasp the basics of each of these views)

Action list: I found that skimming a list of actions represented as text (`ADD_TODO`, `EDIT_TODO`, etc.) required a lot of conscious reading effort. Instead, by choosing a colorful symbol for each action in the app, we can take advantage of pre-attentive processing to more quickly understand what actions have occurred in the execution trace. In this case I chose symbols for all the actions in TodoMVC; more generally, a programmer could specify a meaningful symbol for every action in their application. In some cases it might be difficult to choose meaningful and different symbols for all actions; falling back to random symbols or colored dots could work as well. In using this tool I've found that the symbolic action list makes it far easier to find a point in an execution trace that I'm looking for.

Collection graph: This visualization represents the contents of a collection with a series of vertically stacked dots. The size of the cluster of dots provides a rough sense of collection size, and the programmer can more carefully examine the view to get an exact count.

Each dot has a color encoding for some attribute of the collection element; in this case I've chosen to color the dots by whether the todo is completed or not. Another available option is to color the dots by identity—each unique element gets its own color.

I originally represented the list of todos with a line graph showing its length, but this view allows us to display an additional dimension of information for each todo. One corresponding weakness of this view is that the size encoding doesn't offer too much information for pre-attentive processing when there are more than a few elements so the relative size change is small. It's not immediately obvious where in the trace the number of todos changed, whereas a line graph makes it more obvious. (One possible improvement would be to only show the dots on a time step where the collection was changed.)

Line graph: This is simply a line graph of some numeric quantity over time. In this context I've used it to visualize quantities like "Number of todos visible". Choosing a y-axis is quite tricky because the full range of values can't be known in advance. In trying out different options and using the tool myself, I decided that viewing relative changes over time was most important—generally I'm looking for things like "when did the number of todos go down?". Therefore, I let each graph scale to the current range of values and don't even show a y-axis label—I'm not aiming to precisely read numeric values off the graph.

Enum graph: User interfaces commonly have enums / union types, which can take on a small number of predefined values. To represent enum values changing over time, I chose to use both a color and position encoding, as a way of redundantly encoding the information and .

With more time, I'd like to explore many other types of visualizations in addition to these. One particular interest is displaying the entire state of a collection of objects in a single graph.

Prototype Implementation

I implemented a working prototype on top of the existing Redux Dev Tools, which provides substantial infrastructure for inspecting and manipulating the state of a Redux application. My tool is implemented as a Redux Dev Tools "monitor" which can plug in to those existing APIs.

I used the React and Redux frameworks to implement the main skeleton of my system. The graphs are built in a combination of d3 and React. I use d3 for computing scales and positions, and then React for actually rendering out SVGs.

Discussion and Future Work

This work is still an early prototype and there are many opportunities for future work.

Using this system myself, I found that I was able to more quickly get an overall sense of what happened in an execution trace by looking at these visualizations than looking at the existing Redux Dev Tools display. However, I want to gain a clearer understanding of what questions people have when learning about the behavior of a UI, in order to evaluate the usefulness of the system. In particular, I'm curious about general "program understanding" as opposed to targeted debugging. Could this visualization be a useful aid when onboarding someone into a codebase and teaching them how it works?

There's lots of future work to refine the core visualizations further. I haven't yet explored visualizing a complex object in a single graph, or showing strings changing over time. I'd also like to more clearly incorporate Hoffswell et al's taxonomy [5] into this work, evaluating these visualizations in those terms and explicitly extending that taxonomy.

Another area of work is generalizing this system to work with any Redux application. I'd like to explore the programmer experience of creating these visualizations for an existing complex application. How much of that process can be automated? How can we make it easy for the programmer to decide which visualizations would be helpful, and then to actually specify those visualizations? As an initial idea, I imagine that the programmer could specify an arbitrary expression over the Redux state,choose from a predefined list of visualizations for showing the output of that expression, and then add that to the timeline panel in this tool.

Program visualization offers a rich set of possibilities for helping people understand their code better. In this work, I've provided an initial prototype of a system for visualizing the runtime state of Model-View-Update user interfaces, exploiting the natural architecture of these applications to show an abstract picture of code execution over time.

References

[1] Marc H. Brown and Robert Sedgewick. "A System for Algorithm Animation". In: *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '84. New York, NY, USA: Association for Computing Machinery, Jan. 1, 1984, pp. 177–186. ISBN: 978-0-89791-138-2. DOI: 10.1145/800031.808596. URL: http://doi.org/10.1145/800031.808596 (visited on 05/11/2020).

[2] Evan Czaplicki. *The Elm Architecture · An Introduction to Elm*. URL: https://guide.elm-lang.org/architecture/ (visited on 05/11/2020).

[3] Simon Fowler. "Model-View-Update-Communicate: Session Types Meet the Elm Architecture". In: (Jan. 13, 2020). arXiv: 1910.11108 [cs]. URL: http://arxiv.org/abs/1910.11108 (visited on 05/11/2020).

[4] Philip J. Guo. "Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education". In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE '13. Denver, Colorado, USA: Association for Computing Machinery, Mar. 6, 2013, pp. 579–584. ISBN: 978-1-4503-1868-6. DOI: 10.1145/2445196.2445368. URL: http://doi.org/10.1145/2445196.2445368 (visited on 05/12/2020).

[5] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. "Augmenting Code with In Situ Visualizations to Aid Program Understanding". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: Association for Computing Machinery, Apr. 21, 2018, pp. 1–12. ISBN: 978-1-4503-5620-6. DOI: 10.1145/3173574.3174106. URL: http://doi.org/10.1145/3173574.3174106 (visited on 05/11/2020).

[6] Hyeonsu Kang and Philip J. Guo. "Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations". In: *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. UIST '17: The 30th Annual ACM Symposium on User Interface Software and Technology. Québec City QC Canada: ACM, Oct. 20, 2017, pp. 737–745. ISBN: 978-1-4503-4981-9. DOI: 10.1145/3126594.3126632. URL: https://dl.acm.org/doi/10.1145/3126594.3126632 (visited on 05/07/2020).

[7] Saketh Ram Kasibatla. "Seymour: A Live Programming Environment for the Classroom". UCLA, 2018. URL: https://escholarship.org/uc/item/8gx5x6kj (visited on 05/13/2020).

[8] Andrew J. Ko and Brad A. Myers. "Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior". In: *Proceedings of the 2004 Conference on Human Factors in Computing Systems - CHI '04*. The 2004 Conference. Vienna, Austria: ACM Press, 2004, pp. 151–158. ISBN: 978-1-58113-702-6. DOI: 10.1145/985692.985712. URL: http://portal.acm.org/citation.cfm?doid=985692.985712 (visited on 05/07/2020).

[9] Sorin Lerner. "Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 2020. DOI: 10.1145/3313831.3376494.

[10] Tom Lieber, Joel R. Brandt, and Rob C. Miller. “Addressing Misconceptions about Code with Always-on Programming Visualizations”. In: *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems - CHI ’14*. The 32nd Annual ACM Conference. Toronto, Ontario, Canada: ACM Press, 2014, pp. 2481–2490. ISBN: 978-1-4503-2473-1. DOI: 10.1145/2556288.2557409. URL: http://dl.acm.org/citation.cfm?doid=2556288.2557409 (visited on 05/11/2020).

[11] Josh Pollock et al. “Theia: Automatically Generating Correct Program State Visualizations”. In: *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E - SPLASH-E 2019*. The 2019 ACM SIGPLAN Symposium. Athens, Greece: ACM Press, 2019, pp. 46–56. ISBN: 978-1-4503-6989-3. DOI: 10.1145/3358711.3361625. URL: http://dl.acm.org/citation.cfm?doid=3358711.3361625 (visited on 01/28/2020).

[12] *Redux - A Predictable State Container for JavaScript Apps. | Redux*. URL: https://redux.js.org/ (visited on 05/13/2020).

[13] Steven P. Reiss. “JIVE: Visualizing Java in Action Demonstration Description”. In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE ’03. Portland, Oregon: IEEE Computer Society, May 3, 2003, pp. 820–821. ISBN: 978-0-7695-1877-0.

[14] Steven P. Reiss. “Visual Representations of Executing Programs”. In: *J. Vis. Lang. Comput.* (2007). DOI: 10.1016/j.jvlc.2007.01.003.

[15] Steven P. Reiss and Manos Renieris. “Jove: Java as It Happens”. In: *Proceedings of the 2005 ACM Symposium on Software Visualization - SoftVis ’05*. The 2005 ACM Symposium. St. Louis, Missouri: ACM Press, 2005, p. 115. ISBN: 978-1-59593-073-6. DOI: 10.1145/1056018.1056034. URL: http://portal.acm.org/citation.cfm?doid=1056018.1056034 (visited on 05/11/2020).

[16] John T. Stasko. “Tango: A Framework and System for Algorithm Animation”. In: *Computer* 23.9 (Sept. 1, 1990), pp. 27–39. ISSN: 0018-9162. DOI: 10.1109/2.58216. URL: http://doi.org/10.1109/2.58216 (visited on 05/11/2020).

[17] Bret Victor. *Learnable Programming*. URL: http://worrydream.com/LearnableProgramming/ (visited on 04/28/2020).