Probe Log: Visualizing the Control Flow of Babylonian Programming

Eva Krebs
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
eva.krebs@hpi.uni-potsdam.de

Patrick Rein
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Joana Bergsiek
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
joana.bergsiek@hpi.uni-potsdam.de

Lina Urban
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
lina.urban@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
robert.hirschfeld@hpi.uni-potsdam.de

ABSTRACT

Code itself is abstract, which makes it often difficult to understand – sometimes even by the programmers that wrote it. When working with or thinking about code, programmers thus often resort to concrete values and execution traces to make the abstract more tangible. Such approaches like exploration scripts in a workspace or unit tests of a test suite are already very helpful, but still lack a convenient conceptual and technical integration into core development tools, leaving such examples and the code they refer to too far apart.

Example-based programming like Babylonian Programming aims at offering the benefits of concrete, live examples directly in program editors, interleaved with the code it supports, to shorten feedback loops and reduce the need for context switches coming with changing tools.

However, Babylonian Programming and its tools currently focus on a local perspective on code exploration, but do not yet extend to messages sent outside a particular unit of code and with that do not yet directly support feedback on more dynamic properties of a running program / system.

We developed Probe Log, a Babylonian Programming tool that extends the benefits of example-based programming to scenarios that span across multiple procedures. It provides a linear view on the dynamics of evolving examples beyond a local perspective.

CCS CONCEPTS

• **Software and its engineering → Development frameworks and environments**.

<Programming>'23 Companion, March 13–17, 2023, Tokyo, Japan

ACM ISBN 979-8-4007-0755-1/23/03.
https://doi.org/10.1145/3594671.3594679

KEYWORDS

live programming, exploratory programming, example-based programming, babylonian programming, examples, squeak, smalltalk

ACM Reference Format:
Eva Krebs, Patrick Rein, Joana Bergsiek, Lina Urban, and Robert Hirschfeld. 2023. Probe Log: Visualizing the Control Flow of Babylonian Programming. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming (<Programming>'23 Companion), March 13–17, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3594671.3594679

1 INTRODUCTION

While code itself is abstract, developers often think about it in relation to concrete values. This can happen in several ways. A programmer may run a program and observe the concrete live state. A programmer could look at trace data shared by users or other developers. A programmer might look at an existing test base.

While there are many more ways in which developers may make use of more concrete data for abstract code, all ways aim to provide concrete feedback through example-like concepts. Accessing these examples often requires some sort of context switch such as switching to a browser and searching for a fitting tutorial. It might also require the additional mental mapping of concrete values to code if the example can only be found outside the programming environment. Babylonian Programming aims to provide dynamic concrete feedback as close to the source as possible. This mitigates having to switch tools or mapping values mentally. Programmers can add and inspect examples directly in the code editor itself.

However, current Babylonian Programming tools are focused on a local perspective of a small part of the system, on a micro-view. Yet many programs consist of an interplay of several procedures. Current tools would at least require several context switches and at worst keeping in mind all relevant parts of the system as part of a complex mental model to get feedback on the behavior of multi-procedure programs or systems.

To provide feedback on multi-procedure programs without additional cost such as context switches, we developed an extension for the Babylonian Programming system Babylonian/S. The extension lets developers navigate the control flow of a given example.

We will first introduce Babylonian Programming, a form of example-based programming, and the limit of its local-perspective tools in section 2. We then describe the tool we created to address this in section 3, how exactly it can be used in a concrete scenario in section 4, and how it is implemented in section 5. We briefly outline related work such as other example-based programming tools and other tools that address the limits of local-perspective tooling in section 6. The paper will end with a conclusion and possible future work in section 7.

2 BABYLONIAN PROGRAMMING

The Babylonian Programming system is an Example-based Live Programming (ELP) system that allows for instant and ubiquitous access to dynamic information. This dynamic feedback is provided through two core concepts: *examples* and example-based widgets. Examples provide all the information necessary to invoke the function or method, including the arguments and in the case of a method, a receiver. Further, programmers can designate for which program elements they would like to see dynamic information by adding widgets to expressions in code. If at least one example is available, these widgets can provide dynamic feedback based on the example. The most common type of widget is called a *probe*. A probe traces the values of the selected expression during the execution of an example and the corresponding UI widget displays the value directly within the editor. In combination, examples and probes allow for instant access to dynamic information, as examples make every function or method executable in the system and thereby remove the need for lengthy executions of the whole program to get dynamic information. Through probes, programmers can get ubiquitous feedback on dynamic information right within the code editor. By placing probes on any expression in the program, they can directly inspect and explore dynamic state at this point in the execution.

There are multiple implementations of Babylonian Programming. The first implementation of Babylonian Programming was written in JavaScript in the live programming environment Lively4 [9, 12]. This version first introduced the main concepts of Babylonian-style Programming including examples and widgets such as probes. Programmers could add examples and widgets through and as dedicated-UI elements.

Another implementation was created as part of an example-based live programming plug-in for Visual Studio [11]. The plug-in was created for polyglot programming, which allows programmers to use and combine multiple programming languages. To support polyglot programming, core Babylonian features were implemented in a language-agnostic way. In contrast to other Babylonian Programming environments, examples and probes were added through code comments instead of actual UI-based widgets.

The tool described in this paper was implemented for another Babylonian Programming implementation, Babylonian/S.

2.1 Babylonian/S

Babylonian/S is Babylonian Programming system written in Squeak/Smalltalk [2, 4, 13]. Squeak/Smalltalk is a self-supporting live programming environment. Since it is written in a live programming environment, which aim to provide immediate feedback to enhance explorability and comprehensibility, Babylonian/S tools also provide live programming features [14, 15].

In Babylonian/S, examples are always defined for a given method. There are several types of examples, such as method examples or script examples. Method examples for instance require developers to provide concrete values for receiver and arguments that can be used to invoke the given method, while script examples require a script that invokes the given method directly or indirectly.

Programmers can attach probes and other widgets to expressions in the code. Probes can visualize the return value of the selected expression. Per default, the value is represented by text, but other visualizations such as images for visual objects is possible. Other widgets for instance include replacements, which let programmers replace an expression with a different expression or value, and assertions, which can assert certain properties of the expression. The probes and other widgets can offer a local perspective on a problem context as one code browser can only display the probes of single method and does not provide additional control flow information.

Examples are added through a UI element that is always present at the top of a method while probes and other dynamic widgets are added on demand by the programmer. A code browser with examples and probes can be seen in Figure 1.

2.2 Limits of a Local-Perspective

Babylonian/S provides developers with local insights into the program they are working on. This enhances the understanding of a single code location, such as a single expression. However, at the same time, understanding the overall program requires information about the control flow and thus about the relation between different code locations. To investigate this relation, programmers have to reconstruct the control flow from isolated observations at single locations. Further, even when using an Example-based Live Programming (ELP) environment, programmers have to switch between the different locations. Both these challenges go against the original goals of ELP environments, which are to improve program comprehension through immediate access to dynamic information.

The following scenario illustrates these challenges of the local perspective. A team of programmers wants to work on the text rendering infrastructure of the Squeak/Smalltalk system. Therefore, they want to understand how a text renderer handles a change in font in a rich text.

The text rendering infrastructure in Squeak/Smalltalk is based on Text objects which consist of a string and an array that contains styling information for each character in the string. The rendering itself is done by CharacterScanner classes, which process each character in a text and apply the styling information according to their rendering method. To understand how the change in font takes place, the programmers look at the CharacterScanner>>#setFont method, which is called whenever a change in font occurs, which can be seen in Listing 1.

To get a first impression of how the font changes take place, the programmers use a script example to trigger the rendering of a text that uses two fonts, which can be seen in Listing 2, and put a probe on the first read of the font instance variable in CharacterScanner >>#setFont, which is illustrated in Figure 2. The probe shows six recorded objects, the nil object followed by a font object, followed

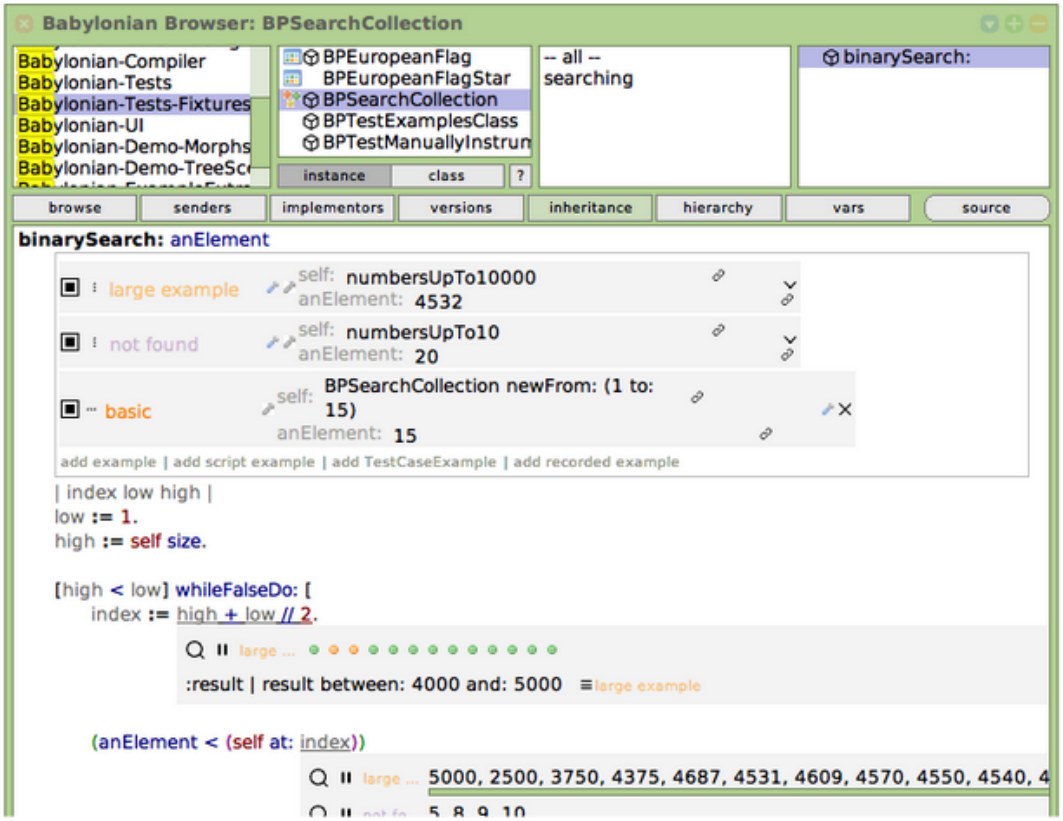

Figure 1: A screenshot of Babylonian/S. Examples are defined at the top of the method while probes are added in the code to visualize it.

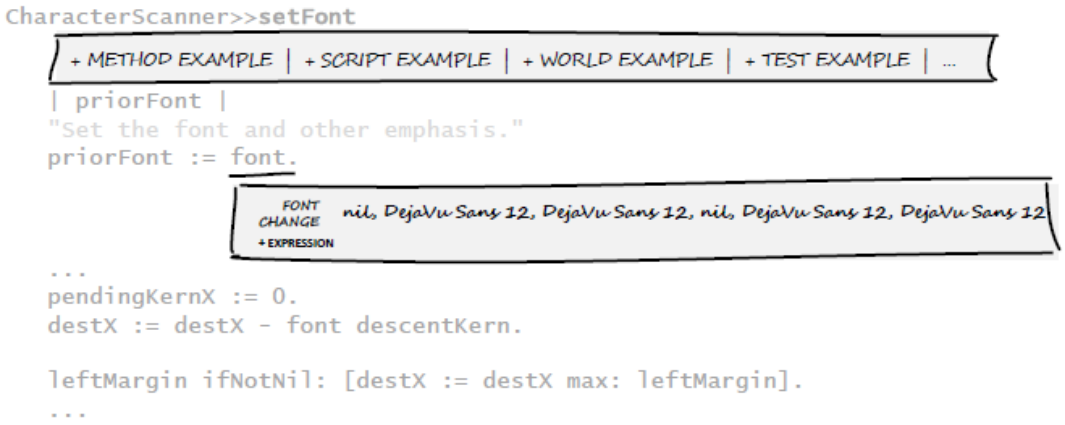

Figure 2: Sketch of a situation in which the local perspective provided by probes does not suffice for programmers to understand the observations. The six objects in the probe are surprising, as the used example only implies two executions of the method.

by two repetitions of the same pattern. The programmers are surprised, as they assumed that there should only be two executions of the CharacterScanner>>#setFont method corresponding to the two different fonts in the rendered text.As the font instance variable is likely uninitialized at first, it is understandable that the nil object is observed followed by a font object. However, it is unclear to the programmers why there are two subsequent repetitions of this pattern, starting with the second time the nil object was recorded.

Listing 1: The setFont method of the CharacterScanner class.

```
1  CharacterScanner>>setFont
2    | priorFont |
3    "Set the font and other emphases."
4    priorFont := font.
5    ...
```

Listing 2: A small script that is used as an entry point for debugging the rendering loop.

```
1  text := 'abc def' asText.
2  text
3    addAttribute: TextFontChange font2
4    from: 5
5    to: 7.
6  "The following triggers a text rendering."
7  text asMorph imageForm.
```

At this point, the local perspective through the probes does not help programmers any further in investigating this behavior. Instead, they would need a "zoomed-out" perspective, showing them what else happened throughout the example execution and how the different recordings of the probe relate to the overall example execution.

3 PROBE LOG

Probe Log offers a new view of a Babylonian example that spans across multiple procedures. It contains a linear view of all probes of a given example that lets programmers inspect them ordered chronologically. This may act as a sort of macro-view of the probed system, enabling programmers to see information and relations beyond the local-perspective.

In order to provide easier access to multi-procedure dynamic feedback, the Probe Log can show a linear list of recorded objects directly in code browser itself, see Figure 3. It only uses information that can be gained directly from a given example's probes and displays this information in order. Since we get all information that probes have access to, the Probe Log does not only display text. The Probe Log retains all reflection features of the probes such as inspecting the contained values. It also contains links to the code its values originate from. Also, just as the examples and probes are updated live when the system changes occur, the Probe Log also reflects the live updates of its example.

Lanes. The Probe Log can display the information of mutiple examples. Each examples is visualized in its own lane. A lane contains a list of all probes and their values of a given example trace. To enhance readability, all values are displayed with the probed expression, or if provided a label, of the probe they originate from. Since large examples may contain several probes and traced values, probes can be hidden if needed.

Log Lines. In order to provide more context on the execution of a given probe, the Probe Log contains Probe Log Lines in each example lane. Probe Log Lines are a flame graph that visualize the execution context of a given probe. There is one vertical bar, or log line, for each context in the stack. The bars are ordered from left to right, the innermost context being on the right. While this is not a complete debugging stack, it provides a general idea of when and why a probe received values. Also, as programmers may mostly be interested in methods containing probes, the context bars of methods that contain a probe are highlighted with a color specific to each probe. Besides examining the visualization, programmers can also use the flame graph to get detailed information and navigate the code. Programmers can explore the flame graph by hovering over context bars, which are then highlighted to show the lifetime of the context and shows the name of the method executed in this context. Programmers can also click on the context bars to open the method in the code editor. Finally, each line includes a pause button that when clicked opens a step-wise debugger on the example execution at the time of the recording of that object.

4 USING PROBE LOG

Probe Log may be used to determine why the probe in the method CharacterScanner>>#setFont(2.2) recorded so many objects. First, one may want to roughly determine when during the text rendering in DisplayScanner these objects were recorded, and thus place a probe at the beginning of DisplayScanner>>#displayLine: offset : leftInRun :, which can be assumed to be the first text rendering method executed.

We then open the Probe Log to understand why CharacterScanner >>#setFont is executed so often, see Figure 4. The Probe Log shows all recorded objects in probes in chronological order. To orient themselves, programmers may look for the probe that was placed in the top-level text-rendering method by looking at the labels at the top of each line in the Probe Log. To their surprise, they find the first line of the probe not at the top of the Probe Log, but in the middle.

By looking at the stack visualization on the left, they realize that only the last two objects were recorded as a result of the text rendering method. The visualization shows this, as the red vertical line represents the top-level text rendering method, which is highlighted because of the probe in this method. The stack visualization of the first group of two objects does not include the vertical red line and thus was recorded in executions of CharacterScanner>>#setFont that were initiated by some other behavior.

In order to determine the behavior that initiated the suspect CharacterScanner>>#setFont executions, they investigate the first recorded value in the Probe Log. They start hovering with their mouse cursor over the area of the stack visualization in the first line. They do so from right to left to follow the sender chain. In one of the first stack frames, they find a method execution on a different scanner subclass, named CompositionScanner. They click on the stack frame of the method CompositionScanner>>#composeFrom: inRectangle: to open it in the browser and continue investigating its purpose.

5 PROBE LOG IMPLEMENTATION

The Probe Log is shown directly within the code browser as a new side panel. All visual elements are based on the Morphic UI system of Squeak/Smalltalk. Multiple examples can be added to the Probe

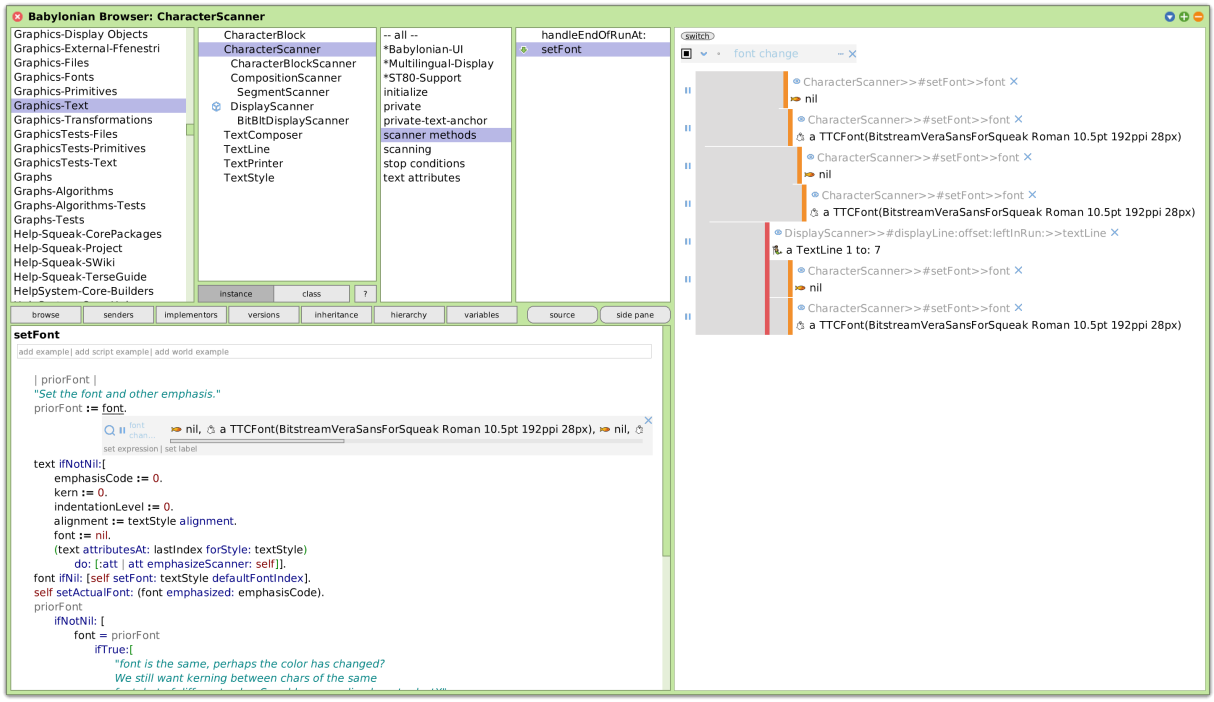

Figure 3: A screenshot of the Probe Log as part of a code browser in Babylonian/S. The right panel contains the Probe Log while the panels on the left contain the other code browser elements such as a code editor for a selected method.

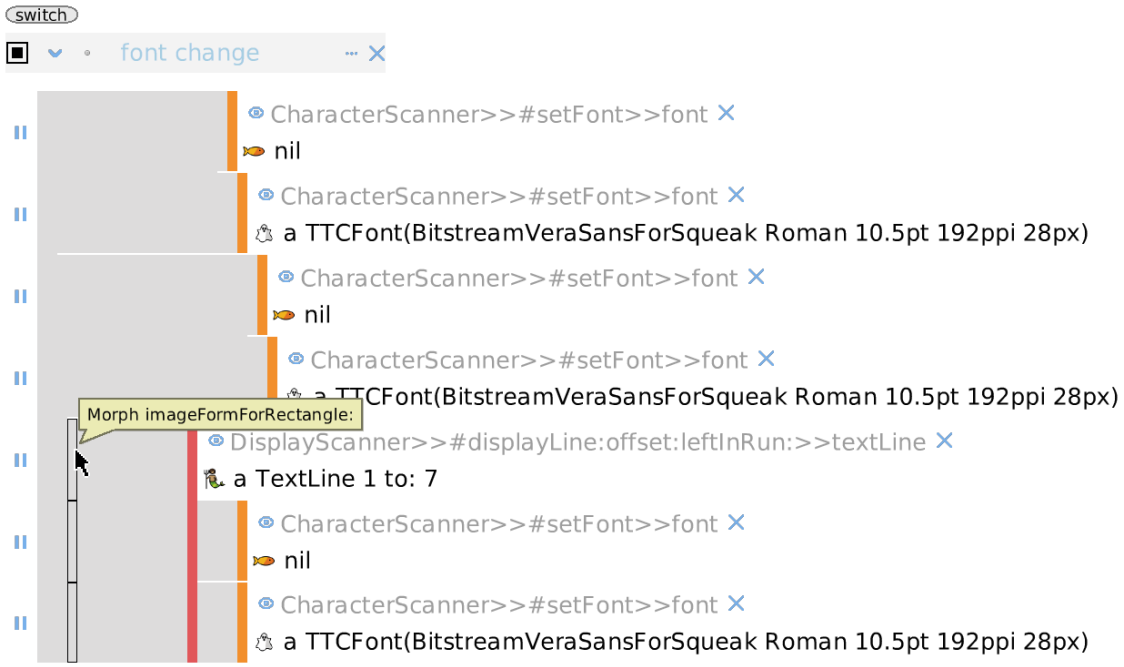

Figure 4: A screenshot of the Probe Log implemented in Babylonian/S. The mouse cursor hovers over a stack frame of the third group of recordings of font objects.

Log, each displayed in its own lane. The example information in a lane is then visualized vertically on the Probe Log lines.

The Probe Log builds on the example infrastructure already present in Babylonian/S. Examples in Babylonian/S collect trace data when executed [13]. This trace data is visualized by the Probe Log.

The Probe Log requires the following information to support its features: the recorded object, the recording probe, the stack at the time of the recording, and a timestamp for the time of recording within the timeframe of the example execution. The basic Babylonian/S tracing infrastructure already provides the recorded object and the recording probe. It also provides a basic form of the stack at the time of the recording in the form of an array of the identity hashes of the active stack frames. This however is not sufficient for showing the flame graph and we extended the trace information with the method selector and the class containing the executed method. Further, the Probe Log requires a timestamp to sort the recorded objects within the lanes. Therefore, we added a timestamp for the time of recording in the form of a simple sequence number that is incremented on every recording.

To support the pausing of example executions at the time of a recording, we also make use of the timestamp. Therefore, we extended the basic tracing infrastructure with a tracing mode in which the example execution is suspended as soon as a specified sequence number is generated. Now, when programmers click the pause button, we pass the sequence number belonging to that log line to the tracing infrastructure and start another example execution. This implementation assumes that the example executions are deterministic, at least in the number and ordering of probe recordings.

6 RELATED WORK

There are several tools and techniques that aim to address the limits of a local perspective similar to the Probe Log.

The first example-based programming environment was *Example-centric Programming* [1]. The example-centric programming environment features a panel next to a code editor that displays concrete values for code based on examples. The new panel shows a trace-like view of the example by showing the system code with concrete values provided by the example execution. This shares similarities with Probe Log, as it also provides an ordered view of the overall system with concrete values. Interestingly, the example-centric programming environment does not include a dedicated mechanism that represents a local perspective but covers fine-grained feedback through the trace panel.

Two environments support combinations of local and cross-cutting perspectives: *Shiranui* [3] and *Seymour* [6]. Both include a basic overall trace view by highlighting the statements that have actually been executed during the execution of an example. In addition, Shiranui allows programmers to select individual intermediate run-time states and see the dynamic slice for that value [3]. Seymour also offers a more advanced view in the form of an icicle plot [8] visualizing the stack over time. Programmers can also use the icicle plot to focus the local perspective on specific stack frames. According to the accounts of applying Seymour to student programs, the icicle plot works well for smaller programs, but does not scale to larger programs, as it is missing relevant information such as the names of called procedures [6].

In its basic structure, the Probe Log is also similar to the trace of outputs in *YinYang* [10]. This trace of outputs displays the results of manually placed print-statements and allows programmers to navigate the code using them.

The *Omnicode* environment takes a very different approach to display overall program behavior [5]. Instead of visualizing the control flow, Omnicode visualizes changes to the complete run-time state throughout the whole program execution. As this only works for small programs, Omnicode is primarily designed for beginner programs. A similar perspective is part of *DejaVu* [7] for interactive camera-based programs, but it requires users to explicitly choose which states should be displayed over time.

7 CONCLUSION

Probe Log provides a dynamic view of probes across several procedures directly in the code editor itself. This allows programmers to use Babylonian Programming for problems that go beyond a local perspective. We have demonstrated this with the walktrough of a specific scenario.

Future Work. While the Probe Log sets recorded objects into their chronological context, it is still an incomplete view of the interactions during an example execution. To allow programmers to investigate and explore overall system behavior, another cross-cutting perspective for Babylonian Programming systems could be used such as full call trace views.

Probes share similarities with printf-debugging. Insights gained from Babylonian Programming and probes could be applied to printf-debugging and vice versa.

While there are descriptions of scenarios and walktroughs for Babylonian Programming and its tools, further user studies could be conducted. The studies could be focused on how to further enhance Babylonian Programming tools, on discovering use cases and domains in which they are especially helpful, and how programmers use them.

REFERENCES

[1] Jonathan Edwards. 2004. Example centric programming. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, John M. Vlissides and Douglas C. Schmidt (Eds.). ACM, 124. https://doi.org/10.1145/1028664.1028713

[2] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley.

[3] Tomoki Imai, Hidehiko Masuhara, and Tomoyuki Aotani. 2015. Shiranui: A Live Programming with Support for Unit Testing. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 36–37. https://doi.org/10.1145/2814189.2817268

[4] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan C. Kay. 1997. Back to the Future: The Story of Squeak - A Usable Smalltalk Written in Itself. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97), Atlanta, Georgia, October 5-9, 1997*, Mary E. S. Loomis, Toby Bloom, and A. Michael Berman (Eds.). ACM, 318–326. https://doi.org/10.1145/263698.263754

[5] Hyeonsu Kang and Philip J. Guo. 2017. 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 2017, Quebec City, QC, Canada, October 22 - 25, 2017*, Krzysztof Gajos, Jennifer Mankoff, and Chris Harrison (Eds.). ACM, 737–745. https://doi.org/10.1145/3126594.3126632

[6] Saketh Kasibatla and Alessandro Warth. 2017. *Seymour: Live Programming for the Classroom.* https://harc.github.io/seymour-live2017/ Accessed: 2023-01-19.

[7] Jun Kato, Sean McDirmid, and Xiang Cao. 2012. DejaVu: Integrated Support for Developing Interactive Camera-Based Programs. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology* (Cambridge, Massachusetts, USA) *(UIST '12)*. Association for Computing Machinery, New York, NY, USA, 189–196. https://doi.org/10.1145/2380116.2380142

[8] J. B. Kruskal and J. M. Landwehr. 1983. Icicle Plots: Better Displays for Hierarchical Clustering. *The American Statistician* 37, 2 (1983), 162–168. http://www.jstor.org/stable/2685881

[9] Jens Lincke, Patrick Rein, Stefan Ramson, Robert Hirschfeld, Marcel Taeumel, and Tim Felgentreff. 2017. Designing a live development experience for web-components. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience, PX/17.2, Vancouver, BC, Canada, October 23-27, 2017*, Luke Church, Richard P. Gabriel, Robert Hirschfeld, and Hidehiko Masuhara (Eds.). ACM, 28–35. https://dl.acm.org/citation.cfm?id=3167109

[10] Sean McDirmid. 2013. Usable live programming. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld (Eds.). ACM, 53–62. https://doi.org/10.1145/2509578.2509585

[11] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. 2020. Example-based live programming for everyone: building language-agnostic tools for live programming with LSP and GraalVM. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2020, Virtual, November, 2020.* ACM, 1–17. https://doi.org/10.1145/3426428.3426919

[12] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming - Design and Implementation of an Integration of Live Examples Into General-purpose Source Code. *Art Sci. Eng. Program.* 3, 3 (2019), 9. https://doi.org/10.22152/programming-journal.org/2019/3/9

[13] Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. 2019. Implementing Babylonian/S by Putting Examples Into Contexts. In *Proceedings of the Workshop on Context-oriented Programming - COP '19.* ACM Press. https://doi.org/10.1145/3340671.3343358

[14] Steven L. Tanimoto. 1990. VIVA: A visual language for image processing. *J. Vis. Lang. Comput.* 1, 2 (1990), 127–139. https://doi.org/10.1016/S1045-926X(05)80012-6

[15] Steven L. Tanimoto. 2013. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming, LIVE 2013, San Francisco, California, USA, May 19, 2013*, Brian Burg, Adrian Kuhn, and Chris Parnin (Eds.). IEEE Computer Society, 31–34. https://doi.org/10.1109/LIVE.2013.6617346