µPrintGen: Supporting Workflow Logs Analysis Through Visual Microprint

Sebastian Alfaro
Department of Computer Science
University of Chile
Santiago, Chile
sebastian.alfaro@outlook.cl

Alexandre Bergel
RelationalAI
Bern, Switzerland
alexandre.bergel@relational.ai

Jocelyn Simmonds
Department of Computer Science
University of Chile
Santiago, Chile
jsimmond@dcc.uchile.cl

***Abstract*—A microprint is a visualization that condenses the contents of a text file into a small space. This is done by using colored scaled-down characters or pixels to represent each character of the text. The colors aim to highlight information relevant to the user. We have created µPrintGen, a framework for creating custom microprints by associating colors to rules that specify what is included or excluded from the line, with plain text or regular expressions. At the same time, the user can customize the look and structure of the created microprint for ease of use. We also created µPrintVis, a website for viewing the created microprints that supports various navigation features, like zoom and search. Feedback from the Rust and Julia open-source communities shows that µPrintGen lets users navigate through large log files, while easily highlighting reported errors. Two experts agree on the benefits of µPrintGen, compared to pure textual-based searching features commonly offered by text editors.**

Video URL: https://youtu.be/lJIZHLQdXi0

***Index Terms*—Microprint, workflow analysis, pixel-based visualization.**

I. Introduction

Microprints offer a way of visualizing source code that is commonly used in modern programming environments. Consider VSCode, one of the mainstream programming environments. Each text editor window includes a vertical panel by default, where a representation of the source code, as seen from a far-away point of view, is shown. We refer to this visualization as a *microprint*. Conceptually, each character from the source code is represented as one pixel or scaled-down character in the microprint. Our implementation retains the white space found in the original file for ease of understanding and analysis. Coloring could be employed for a number of reasons, including highlighting method/function definitions, comments, and results of text searches.

Building and executing automated workflows are an essential part of software development nowadays. As part of continuous development and continuous integration (CI/CD) practices, workflows specify tasks that can be automated, thus alleviating practitioners from manual and tedious tasks (e.g., running unit tests and deploying applications). GitHub Actions can execute a workflow upon the occurrence of particular events like git commit or merge. Executing a workflow typically produces large log files, which practitioners can access from CI/CD systems as a single text file. Searching in such logs typically involves using the mouse to scroll, as well as using textual pattern searching.

This paper is about using microprints to help practitioners extract knowledge from log files, using the CI/CD scenario as motivation. We have developed three complementary tools:

- *µPrintGen*, a framework for building context-specific microprints from text files, where the color highlight rules are specified in a separate configuration file.
- *µPrintVis*, a tool for visualizing microprints;
- *GHµPrintGen*, a GitHub Action that uses µPrintGen to generate a microprint from a GitHub Actions workflow, and generates a link to easily visualize it with µPrintVis.

The hypothesis we are exploring is that microprints significantly help practitioners (i) extract knowledge from a workflow log file and (ii) navigate between components.

II. Using Microprints to Visualize Workflows

A. CI/CD workflows

Automated workflows are an essential part of software development. Recommended as part of standard CI/CD practices [1], workflows are used to make CI/CD tasks easy to run automatically and regularly. Examples of CI/CD tasks include unit test execution, automatic deployment, and binary building. Logs produced by the execution of those CI/CD tasks are then made available to the end user for manual inspection. This manual inspection is often necessary to understand the cause of a failure, debug the workflow, or simply assess whether the workflow was executed as expected.

In other words, automated workflows are a series of jobs, run in succession or in parallel, that contribute to a particular CI/CD task. The output of each job or step in the automated workflow is logged for the user. Logs are text reports that inform the user about the status of the work that was carried out, errors, or general information about the process.

There are several CI/CD platforms available, and although they have differences, one aspect that all popular platforms have in common is a lack of adequate tooling to let end-users easily process workflow logs. All these platforms offer is text-searching through a basic search box, so end-users usually need to resort to reading log contents in depth [2]. Treating logs as simple text files makes it hard for end-users to derive relevant patterns from the log contents.

At the same time, CI/CD processes are commonly run multiple times with small differences in the related application base code or in the workflow definition. Detecting anomalies in a process that can have tiny differences between runs makes it a taxing job to analyze logs: errors or anomalies can go by silently without alerting the people involved.

B. Microprints

Robbes et al. [3] define microprints as "pixel-based character-to-pixel representations of methods, enriched with semantical information mapped on nominal colors". Bacher et al. [4] propose the notion of a code-map metaphor as "the mapping of source code to a zoomed-out representation, either by the use of pixels, pixel lines, or a scaled-down representation of text". In the context of helping visualize large log files, we merge both these definitions into the following one: *zoomable and navigable pixel-based character-to-pixel representation enriched with semantical information related to workflow mapped on nominal colors*.

Fig. 1: Overview of our approach.

Figure 1 shows our approach. µPrintGen first takes as input a log file and a configuration file and generates a set of SVG descriptions as output. µPrintVis is then used to visualize the corresponding microprint.

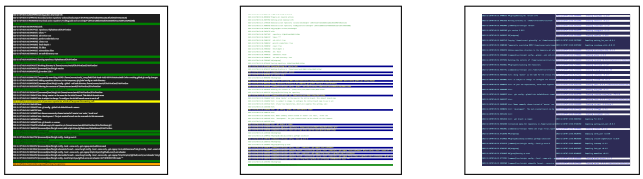
µPrintGen is a Python package/console command that creates a microprint representation of any text file, using a configuration file and saving it to an SVG file. The microprint representation generated uses a scaled-down font of each character in the text file and preserves the original file indentation and spacing. This is an important aspect of this component since this direct mapping between the text file and the microprint improves navigation.

µPrintGen generates the microprint by scanning each line of the input log file and assessing if the line meets the criteria of any of the rules set in the configuration file (text search or regex match). If a rule is matched, the background and text colors of that line are set to the ones defined in the configuration file for that rule. By highlighting important lines with different colors, the small size of the microprint lets users identify visual patterns and possibly relevant lines in the log.

Users can customize almost all visual aspects of a microprint by creating a configuration file. This includes defining the scale factor, the amount of vertical spacing, the microprint width, the maximum microprint height, the number of columns, the space between these columns, the font color, and the font family to be used.

As configuration files offer a high level of customization, not only in how a microprint appears, but also in what information is highlighted. Two microprints generated by two projects from different communities can look entirely different from one another. This is an important feature of our tool because different projects have different information that needs to be highlighted. For example, some CI/CD workflows have tests, some do not; some projects need to search for specific errors, others want to highlight any error, etc.

For example, the microprints shown in Fig. 2 have been created from the same log file but with different configurations. They have different fonts, background colors, text colors, and column widths and the third one even shows the text in two columns instead of one.



(a) (b) (c)

Fig. 2: Different microprints generated from the same log file.

Here is the JSON configuration file with the color mapping we defined for our proof-of-concept example from the Rust community.

```
{
  "scale": 2,
  "vertical_spacing": 1.4,
  "microprint_width": 140,
  "default_colors": {
    "background_color": "rgb(30, 30, 30)",
    "text_color": "white"
  },
  "line_rules": [
    {
      "includes": [
        "(?:^| )(?:error|panicked|failed|
          stacktrace)(?:$| |:)"
      ],
      "excludes": [
        "installing"
      ],
      "text_color": "white",
      "background_color": "#910404"
    }, {
      "includes": [
        "(?:^|\\W)warning(?:$|\\W)"
      ],
      "text_color": "black",
      "background_color": "#dfdf07"
    }, {
      "includes": [
        "(?:^|\\W)test result(?:$|\\W)"
      ],
      "text_color": "black",
      "background_color": "#d0e6c9"
    }, {
      "includes": [
        "(?:^|\\W)successfully(?:$|\\W)"
      ],
      "text_color": "white",
      "background_color": "rgb(38, 162, 105)"
    }
  ],
  "font–family": "monospace, monospace"
}
```

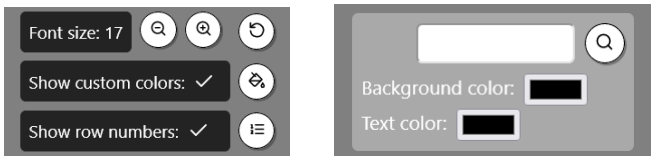
Listing 1: JSON configuration file for Rust proof-of-concept.

These rules highlight erroneous messages, warnings, test results, and whether the build was successful. The microprint in Fig. 4 was generated using these specific rules. The entire microprint can be seen on the website **https:/bit.ly/Microprint-Example-Rust** (albeit with slight modifications to avoid false positives).

µPrintVis is the visualization tool we designed to help users read and navigate through logs, aided by their microprint representations. This tool loads the microprint generated by µPrintGen and creates a full-size representation of it, using the same highlighting, colors, and font size. That way, the user can use the microprint representation of the logs to locate areas of interest, and quickly navigate to these lines by clicking the microprint. At the same time, the tool offers a search tool to highlight patterns using custom colors in real time.

The user can also generate a microprint by uploading a text file and a JSON configuration file to µPrintVis, like the one shown in Listing 1. In this case, µPrintGen is first used to generate the microprint and then loaded in µPrintVis for easier traversal and analysis. The uploaded files are only temporarily used to generate the microprint, they are not saved to our server.

The microprint is located to the right of the full-size text and can be used in different ways to navigate the document. If the user clicks on the microprint, the full-size page scrolls automatically to the corresponding line. When the user hovers over the microprint, a semi-transparent square appears. This square denotes the visible area of the page in the microprint and can be dragged to scroll the full-size page.



(a) View customization settings (b) Search box

Fig. 3: Parts of µPrintVis settings menu

The visualization tool has a settings menu that gives the user the ability to customize the viewing experience of the text and microprint, among other things. For example, the options shown in Fig. 3a) let the user (i) increase/decrease the font size of the full-size text; (ii) toggle the color highlights of the full-size text, which can make it easier to read log messages once they have been found through the microprint and (iii) show/hide row numbers. The search box shown in Fig. 3b lets the user search for a phrase in the microprint. What makes this search function different from the browser search function is that the user can specify the background and text colors for the search results, so all the lines that match the phrase are highlighted with those colors. This can help the user define new configuration rules, or find patterns they had not yet seen.

Finally, the user can download the generated microprint to analyze it later. This is an SVG file that has the entire text in microprint format, with the highlights and other visual cues defined by the configuration file. This file can be directly uploaded to µPrintVis for visualization, the user does not need to regenerate the microprint in order to visualize it.

C. Integration with GitHub Actions

GHµPrintGen is a GitHub action that makes use of µPrintGen to generate microprints as part of an automated GitHub CI/CD workflow. Here, jobs are sets of steps, which can be shell scripts or actions to be executed. GHµPrintGen is one such an action.

GHµPrintGen is designed to automatically generate a microprint of the logs generated by a job inside a GitHub workflow. For that, it needs to be added as a step inside a workflow. Ideally, inside a job whose sole purpose is to generate the microprint of another, due to GitHub limitations.

In order to facilitate the analysis of the resulting microprint, by default this action will create a Markdown file that includes a link to µPrintVis, with the microprint already loaded (this behavior can be turned off in the GitHub action options). Otherwise, the user would need to download the generated microprint and upload it µPrintVis each time they want to analyze the log file. The Markdown file is saved to the user's computer. We use Markdown because these can include links to other Github pages.

III. Preliminary Evaluation

To gather initial feedback about the usefulness of our tool suite, we looked for projects that regularly use CI/CD workflows that could benefit from viewing microprints of their CI/CD logs. Both the Julia and Rust programming language projects meet these requirements.

We created a microprint for each project from existing logs and presented them to the corresponding community, using existing project communication channels: a Slack group for Julia and a Zulip channel for Rust. We got feedback from two users, one from each community. Even though the sample size is small, other members of the project expressed interest in our tool, reacting positively to our posts.

The initial feedback we received is promising with respect to two aspects. First, both users expressed interest in the tool. One even said that the tool was something they had "wanted for a long time". The second promising aspect of the feedback is regarding the perceived usefulness of the tool. The Julia project member said that: "[I] have to sit through long log files to find one line that shows me a bug, and I think this tool would be very helpful". The Rust project member found a real bug in the workflow that needed to be fixed, which we show in Fig. 4. This error should have terminated the workflow but did not. It had not been found through manual inspection of the log file, since it is thousands of lines long.

These are two real examples of how our tool suite can be used by developers who work with non-trivial workflows, generating large log files. It facilitates the analysis and monitoring of the workflows, highlighting possibly relevant log lines. Also, the highlighting rules can be customized to the specific needs of each project and what they consider important.

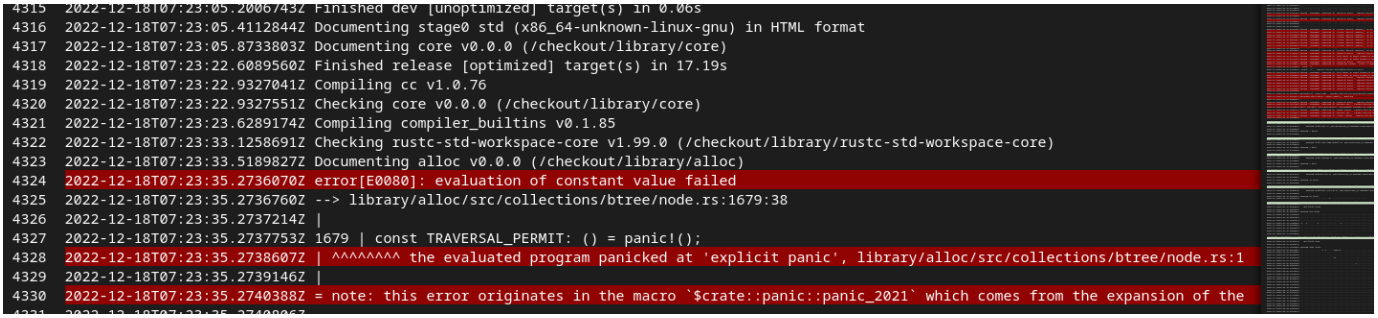


Fig. 4: Example of a real bug found in the Rust CI/CD workflow. The rules were set to highlight lines that contain error information and lines that contain the word "successfully". The microprint helped a community member find this error in a 4.5k line log file, by looking at a compact representation of the entire log.

(a) Sublime: minimap (b) Kate: scrollbar minimap (c) VSCode: code minimap

Fig. 5: Microprint examples from existing tools

IV. Related Work

The idea of microprint has been given many different names, like code-map, minimap, and microprint, and while they all share the principle of representing text in a compact format, the exact implementations differ from each other. Here we present existing implementations, and we discuss how they differ from our solution.

The code map metaphor [4]. (1992-2015). Bacher et al. conducted a systematic literature review about the use of the "code-map metaphor" (what we call microprint in this context) for software visualization. The primary research question the paper aims to answer is: "*How is the code-map metaphor employed by existing software visualizations and what evidence exists of its usefulness?*"

The authors analyzed 29 primary studies, which together describe 21 software development tools that make use of the code-map metaphor for visualization (published between 1992 and 2015). The authors used the following dimensions to guide the data extraction process: (i) Task: why this visualization was needed; (ii) Audience: who would use this visualization; (iii) Target: which aspects of the source code are visualized; (iv) Representation: how accurate is the visualization; (v) Medium: how is the visualization shown; and (vi) Evidence: was the visualization effective.

The paper concludes that the code-map metaphor is "widely perceived to be useful for software development" because it serves as a natural mapping from the source code to a visual representation. Also, the qualitative validations and user experiences reported in the primary studies also align with this vision. However, they also reported there was a lack of sizable quantitative evaluations of these visualizations, making the claim of the usefulness of the visualization for software development weak.

All of the tools presented by the primary studies have fixed rules and designs. This means that they only highlight a set number of elements, depending on the goal of the visualization. For example, one tool highlights source code syntax errors, while another uses different colors to show the age of different parts of the source code, and another uses colors to highlight developer activity. At the same time, the way this information is shown is fixed for each tool: users cannot customize the colors, font sizes, width of the visualization, etc.

We address these limitations by giving the user the ability to customize not only what gets highlighted and with what colors, but also how the whole visualization gets rendered; from font families and font sizes to the number of columns and amount of vertical spacing between rows of the complete microprint representation.

Sublime Text. [5] [5a] (2010) Sublime is a text editor that was presented for the first time in 2007 [6], and in 2010 it began to include microprints [7]. To our knowledge, it has not been explicitly named anywhere except in the configuration files, where it is referred to as a minimap.

Here, the minimap is shown to the right of the text area, with each character of the text as a scaled-down font representation of the character in the minimap. A semi-transparent square appears in the minimap when the mouse hovers over it, which shows the visible area in the text at that current moment. Clicking in the square and dragging it up and down scrolls the whole document. Also, clicking anywhere in the minimap scrolls the document to the corresponding part.

In terms of customization, the user cannot customize the rules of highlighting for the microprint, but can customize what gets highlighted by creating a custom theme for Sublime.

The custom theme can have a set of rules that uses scopes, which define what gets highlighted and how. It permits the highlighting of specific types of words, like variables, strings, numbers, etc. However, the user cannot specify specific words to highlight, like a phrase in a line.

Compared to our implementation, the customization in Sublime is on the one hand more powerful, since it permits highlighting of words and not only lines, but it cannot highlight based on the content of the word. Also, the user needs to create an entire theme for the application for each use case they may have, which limits it in the context of analyzing different logs from different projects.

Kate. [8] [5b] (2014) The text editor Kate uses a similar implementation of the concept, calling it scrollbar minimap instead. It only shows a pixel representation of the text, not the actual words in a smaller font. As for visualization, it also uses the area highlights to show the visible area of the document, and the user can scroll the document by dragging it. This highlight area is always visible, unlike the Visual Studio Code implementation, which disappears until the user moves the cursor over the minimap.

An important difference to our tool is that the Kate scrollbar minimap does not move proportionally to the scroll of the full-sized document. The scrollbar minimap is exactly the same height as the text editor window. So no matter the amount of text in the file or the height of the window, the scrollbar minimap never scrolls. This means that when there is more text in the file, or the window height gets smaller, the size of the text representation inside the scrollbar minimap decreases to make it fit the existing area.

In terms of customization, the only way the user can customize the colors of the minimap is by changing the theme of the whole editor, or by changing the text file type. Kate automatically infers the file type from its extension, and changes what it highlights and in what color. For example, in a Python file, the function definitions will be highlighted in different colors. There is no way for the user to set rules about what should be highlighted and in what colors, apart from the theme and rules associated to the file type. Our implementation gives the user this option.

Visual Studio Code. [9] [5c] (2017) The code editor Visual Studio Code uses a visualization of the contents of the files that correspond to our definition of what a microprint is. Although it is placed in the editor's context, it is called a minimap. It shows a condensed representation of the files, using a small font, to the right-hand side of the screen, by default. It offers some customization options, like showing the minimap to the left of the text file, in one of three sizes, and the user can toggle between showing colored pixels instead of the real characters.

We took great inspiration from its presentation of the microprints, as well as how the user can interact with them. As the user scrolls the page, the microprint scrolls proportionally. At the same time, there is a highlighted semi-transparent area in the microprint that shows the visible area inside the document at any time. That way, the user always knows what part of the file is being shown in the microprint. An added bonus of the highlighted area is that the user can scroll the document by dragging it, which helps with navigating the microprint.

In terms of customization, this implementation is a bit more robust than the others because is backed up by the ability of the editor to have extensions. These extensions can be used to add rules so that the minimap to highlight different things in the source code. Common implementations of these are language-specific linters, that highlight syntax errors and warnings of interest to that specific programming language. On the other hand, personalization of the minimap itself is lackluster, as it only offers three possible sizes and does not allow color or font changes.

V. Conclusion

Microprints for workflows. This paper presented a tool suite intended to alleviate the search and navigation in logs from CI/CD workflows. Our initial and informal evaluation illustrates the benefits of our tool suite as indicated by members of the Julia and Rust programming language communities.

Future work. The biggest limitation of the current implementation of the microprint is that it can only highlight information based on line-by-line regex or simple word search. This is a limitation because even though regex can be powerful, it can be computationally expensive. Also, applying highlighting line by line does not let the tool find blocks of data that could be useful or interesting to the end-user.

As such, we plan to explore different ways to help users highlight useful data. For example, we would like to add configuration rules to aid in the identification of structures in the logs and blocks of text that span multiple lines. We also want to add other ways of testing that a line or multiple lines fit a certain pattern, additionally to regex pattern matching.

Another possible enhancement of the tool is to give the user the possibility of comparing two versions of the text and highlighting the difference between the two. This could help analyze CI/CD workflows by showing a clearer view of the changes introduced between one version and another and for identifying anomalies or unexpected changes.

References

[1] GitHub. (2021) Continuous integration and continuous delivery (ci/cd) fundamentals — github resources. [Online]. Available: https://resources.github.com/ci-cd/

[2] Katalon. (2023) Best 14 ci/cd tools you must know — updated for 2023. [Online]. Available: https://katalon.com/resources-center/blog/ci-cd-tools

[3] R. Robbes, S. Ducasse, and M. Lanza, "Microprints: A pixel-based semantically rich visualization of methods," in *European Smalltalk User Group*, 2005.

[4] I. Bacher, B. Mac Namee, and J. Kelleher. (2017) The code-map metaphor - a review of its use within software visualisations. [Online]. Available: https://arrow.tudublin.ie/scschcomcon/223/

[5] Sublime. (2023) Sublime text - text editing, done right. [Online]. Available: https://www.sublimetext.com/

[6] ——. (2007) November 2007 - news - sublime hq. [Online]. Available: https://www.sublimetext.com/blog/articles/2007/11

[7] ——. (2010) February 2010 - news - sublime hq. [Online]. Available: https://www.sublimetext.com/blog/articles/2010/02

[8] KDE. (2023) Kate - get an edge in editing. [Online]. Available: https://kate-editor.org/

[9] Microsoft. (2023) Visual studio code - code editing. redefined. [Online]. Available: https://code.visualstudio.com/