L-Vis: Visualizing Language-Level Provenance for Program Comprehension

Francis Nguyen and Joseph Wonsil

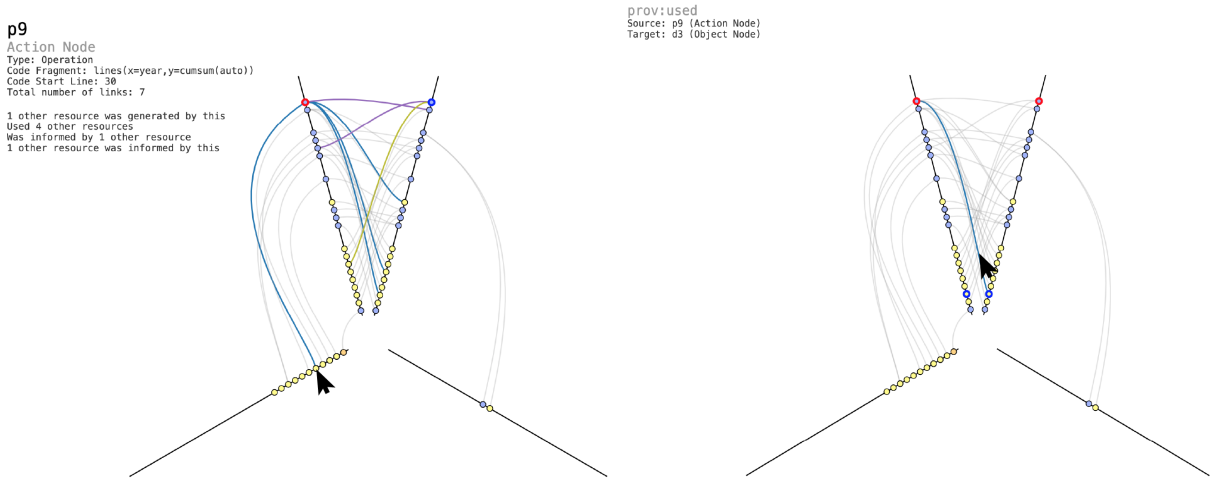

Fig. 1. Two examples of interaction with hive plots in L-Vis. Clicking on a node displays additional information about the selection (left). Users can view related nodes that have relationships to the current node. Hovering over a link between two nodes shows the dependencies between the two (right).

Abstract—There are many situations where collaborators have to understand previously written code; users reviewing their colleague's analysis scripts, an engineer returning to old code personally written long ago, or new researchers inheriting legacy code for maintenance. To aid with program comprehension, provenance, the history of the derivation of an object, is typically captured and often visualized to understand the history of functions and variables throughout the program process. However, the scale of provenance that is collected can overwhelm a person's ability to parse it. We present L-Vis, a language-level provenance (LL-Prov) visualization tool that leverages the hive plot visual idiom, a semantic graph layout visualization, to more accurately discern relationships and structure in scientific analyses. We find that L-Vis is more accurate and efficient than other language-level provenance tools, and users subjectively prefer the experience of L-Vis over alternatives.

Index Terms—Program comprehension, Scientific analysis, Provenance, Network layouts, Hive plots.

1 Introduction

It is a common occurrence where a graduate student inherits convoluted code from a previous, possibly graduated student. Their advisor then asks them to update the code and add new features. To complete the task, this new student has to figure out how the confusing code works, often beginning by constructing a mental model of the code structure. Understanding how programmers interpret new code is the subject of research in program comprehension. Specifically, previous work in program comprehension defines common questions programmers ask or tasks they complete to achieve comprehension [12, 25, 28]. Based on these studies, we hypothesize that *provenance* can be a helpful tool to facilitate program comprehension.

Provenance is the history of how an object came to be in its current state. In computer science, tools collect and use provenance at various scales in different domains. For example, system-level provenance is a record of kernel objects and the relationships between them while application-level provenance is a record of inputs and outputs to programs and the connections between them. Language-level provenance (LL-Prov) is a history of the execution of a program at the source code level. This history includes control flow, values, types of variables, libraries imported, and the connections between all of these [13].

However, it is not easy for even trained users to parse this historical information. This difficulty is in part due to the representation of LL-Prov: a directed acyclic graph (DAG) consisting of multiple nodes and edges. Users need to sift through databases or long files of nodes and edges to make sense of collected provenance. A common solution to this problem is to provide visual assistance through information visualization [10, 27]. Provenance in the form of network data can be directly visualized as a node-link graph.

Naturally, with visualization solutions also come visualization problems. For example, visualizing network data at scale is a huge issue: networks quickly become illegible when displayed on a screen due to the scale of the information [23]. To ensure this data is understandable to a user, visualization designers must carefully judge design trade-offs between filtering, aggregating, and displaying network data. Additionally, designers must consider the problem domain, domain-specific tasks, and users their visualization is designed for. Failure to take into account any one factor can result in an ineffective visualization that

• *Francis Nguyen and Joseph Wonsil are with the University of British Columbia. E-mail: frnguyen — jwonsil@cs.ubc.ca.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

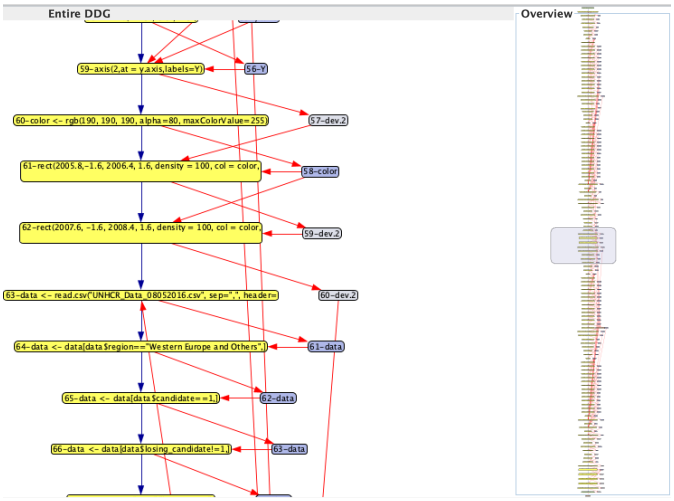

Fig. 2. DDG Explorer can encounter problems with large provenance graphs. This figure shows a real-world analysis visualized by the tool. The graph is linear and only a small portion can be read at a time, and it is near-impossible to infer the data flow from start to finish.

does not fulfill the requirements of its target users. Based on our own examination of the domains of provenance and program comprehension, we created a visualization intended for use by scientists writing data analyses.

We present L-Vis[1], a tool for LL-Prov visualization, designed to assist users in comprehending data analyses written in the R statistical computing language. L-Vis extends a previous reproducibility workflow tool, containR [8], and adds a visualization component to facilitate interpretation of R analyses. We choose to display provenance with a hive plot network layout [17]. Hive plots are a variation of a node-link graph that semantically layout nodes across multiple axes to encode more information into the display. We hypothesize that compared to traditional LL-Prov visualization tools, this visualization idiom more effectively encodes R script structure, adding clarity to relationships between nodes of the provenance graph. We define a set of task-abstractions and requirements for L-Vis by conducting a set of unstructured interviews with programmers of varying experience. We present the results of an informal qualitative study to evaluate L-Vis' utility for program comprehension tasks.

Our contributions are therefore (1) characterization of common tasks necessary for program comprehension when analyzing language-level provenance, (2) L-Vis, a language-level provenance visualization tool that utilizes hive plots to semantically groups nodes according to desired attributes, and (3) a qualitative user study examining the utility of L-Vis for program comprehension.

2 Related Work

We motivate and contextualize L-Vis based on previous work in provenance, network, and workflow visualization.

2.1 Provenance Visualization

The design spaces of prior provenance visualizations are perhaps the most relevant for the design of L-Vis. We examine visualizations tools that depict varying levels of provenance.

DDG Explorer is a visualization tool that targets the same goal of visualizing LL-Prov. This tool is specifically designed for language-level provenance (LL-Prov) generated from the R language [18]. DDG Explorer creates node-link graph visualizations where each node represents executions, variables or other control structures, and edges represent control and dataflow. Similar in spirit but different in scale, VisTrails utilizes a node-link graph, but is designed for application-level provenance [7]. Researchers can use this tool for exploratory analyses where the aim is to visualize scientific workflows, allowing them to keep track of previous steps in their exploration. Both these provenance visualization tools fail for common reasons. For large, complex datasets, the strategies employed by DDG Explorer and VisTrails tend to break down and become increasingly difficult to interpret and derive structure from. Additionally the graph layout algorithms used by both these tools are relatively simple: layout relationships vertically along the canvas, stretching the node-link graph long. This type of layout algorithm can make it easier to see direct relationships between parents and children, however makes it impossible to follow nodes whose connections extend to higher depths. How DDG Explorer handles large provenance graphs from a real-world analysis can be seen in Fig. 2. L-Vis directly addresses this concern, choosing an opinionated way to filter data and leveraging a graph layout algorithm that semantically encodes node position.

ProvThreads captures application-level provenance data and displays user interaction logs to show an analyst's thought process [22]. This tool displays user interactions through colored lines that represent different data topics. Mohseni et al. chose this encoding based on their task abstractions to analyze user workflows and decision points in scientific analysis. However, this encoding is incompatible with L-Vis as the tasks between these tools differ significantly even though provenance is the target for both visualizations.

As a contrast to the traditional node-link graph, InProv is a top-down tool designed for whole-system provenance [5]. InProv leverages radial layouts of time-based hierarchical groupings to improve accuracy and efficiency of finding nodes in addition to helping users understand high level concepts of the provenance. Borkin et al. find that these layouts improve accuracy and efficiency of finding nodes, but visualize system-level provenance instead of LL-Prov. Researchers constructed this tool as an alternative to Orbiter, a node-link graph visualization that suffers from the same scalability issues as DDG Explorer [20]. While InProv slightly improved participants accuracy and efficiency in the quantified tasks in system provenance, we find that these tasks abstractions do not adequately cover the range of tasks necessary for program comprehension. Additionally, while a small majority of participants preferred InProv to a more traditional graph layout (Orbiter), the other subset of participants noted that traditional node-link graphs were more familiar and therefore more preferable.

In our work, we embrace the design lessons of InProv: instead of leveraging a new visual idiom, we extend the traditional node-link graph so that users may feel comfortable with a familiar visual encoding. However, we improve on this layout by encoding additional semantic information in the graph layout.

2.2 Network and Workflow Visualization

The network visualization community has spent extensive time developing effective techniques for drawing large complex networks [1–3, 14]. Some work suggests multiple views of a visualization in order to provide context to a user; a main canvas in addition to zoomable views where a user can investigate further [27]. We found these dashboards to be good points of reference. In particular, ExtraVis is a trace visualization tool that provides two interactive views of large execution traces [9]. One view displays a massive sequence view of the trace and the second produces a "circular bundle" view that hierarchically projects the program's structural entities on a circle. ExtraVis visualizes system-level traces as opposed to LL-Prov. While the tool does answer some questions on granular program comprehension for debugging, it fails to encapsulate high-level program comprehension.

BioFabric [19] is a tool that proposes a novel approach to network visualization. BioFabric is specifically designed for scale in the domain of biological research. BioFabric encodes nodes as one-dimensional horizontal line segments, breaking the notion of the typical node-link graph representation. While BioFabric succeeds in visualizing structure

[1]This project was done as a dual project for CPSC 547 and CPSC 508. The structure of this paper is written to match the layout of 547, but still include some of the work that was performed for 508. Some notes are distributed throughout describing what was done for which course, as well as a section at the end describing this in more detail.

in large scale biological datasets, its approach loses visual effectiveness when representing medium-sized networks or smaller.

While both workflow visualization tools give examples of the visualization design-space of dependency relationships, they both consider trade-offs in design targeting specific scales. ExtraVis does not adequately address the scale issue of provenance data while BioFabric cannot effectively communicate structure in smaller scale networks. While neither tool is designed with LL-Prov in mind, we take into consideration both approaches and combine their philosophies in the design of L-Vis.

3 Data and Task Abstractions

We discuss our data abstraction for the scale of provenance we are using and the tasks we defined through user interviews and program comprehension research.

3.1 Task Abstraction

To derive domain-agnostic requirements of L-Vis, we first identify common tasks from program comprehension literature. Tasks in program comprehension outline requirements that are fulfilled when constructing conceptual models of code. We narrow down the many tasks proposed in program comprehension by conducting unstructured interviews to gather requirements from our target users. We conduct interviews with graduate students who have recently inherited code in the past term, students in introductory programming courses, and professors with a research focus in program comprehension. Our main goal in this process is to identify common tasks and processes users undertook to understand unfamiliar code. A complementary goal was to define requirements that an effective visualization should address.

Program Comprehension Tasks

Extensive work in program comprehension explains cognitive models employed by programmers in solving and accomplishing real-world tasks such as fixing bugs or adding new features.

Storey discuss multiple program comprehension strategies that software engineers employ [29]. These mental models involve top-down, bottom-up, and interactive approaches, where software engineers incrementally make changes to the code base to see how varying inputs to the system change the outputs. Storey notes that program comprehension is not an end goal, but rather a necessary step in achieving another goal. The types of mental models employed by programmer differs between people, so we narrow the scope of L-Vis to prioritize the top-down cognitive strategy. In the top-down strategy, programmers generate assumptions about the structure and end-result of a program and then investigate every subsection of the program with regards to those assumptions.

Sillito et al. improve on the development of these mental models, conducting a set of interviews to determine questions programmers ask themselves when trying to understand a codebase [28]. While these questions can predict the author programmer's mental model, their usage in practice seems impractical as it assumes a experienced programmer's perspective towards program comprehension.

Erdos and Sneed approach program comprehension from a program maintenance perspective. They propose seven questions that need to be answered for a software engineer to successfully maintain a program that is only partially understand [12]. Some of these questions include "Where is a particular subroutine/procedure invoked?", "What are the arguments and results of a function?", and "Where is a particular variable set, used or queried?".

Pacione et al. refine these questions by conducting a literature review of program comprehension, ensuring that their task set constitutes a comprehensive range of typical software comprehension tasks [25]. They construct abstract questions that should be answered for program comprehension such as, "What interactions occur between objects?", "What is the high-level structure/architecture of the software system?" or "How do the high-level components of the software system interact?" While this list of questions covers a large breadth of tasks, most tasks are too broad to apply to use of visualization for program comprehension. To narrow the scope of all the task sets that can apply to visualization in program comprehension, we define a user group and their requirements for a visualization system.

Unstructured User Interviews

We conducted unstructured interviews with participants of varying coding expertise to learn about their data analysis and comprehension workflows. Our participants included introductory programming students, computer science graduate students and a university professor whose research lies at the confluence of software engineering and computer science pedagogy. The interviews lasted approximately 15-60 minutes. The interviews were contextual in the participants work environment where the interviewee described their workflow and tools that they currently used to understand unfamiliar code. We corroborate collective notes from the set of interviews to derive common high-level tasks and themes from the interviews. Despite the range of strategies to build cognitive models of code, similar themes of visualization requirements emerged. All groups believed they could effectively analyze small snippets of code, but needed a larger representation to base their cognitive model on. This reveals the requirement that a high-level context view displaying a summary of available provenance data is necessary. In addition, all participants wanted to be able to examine a specific code snippet's derivation in terms of dependencies and previous calls. A graduate student noted that, "tracing these relationships [between code-snippets] can facilitate hypothesis-driven debugging."

Based on our interviews, we present six high-level tasks that inform the design of L-Vis:

1. *Display affected code if a user changes a variable's value.*
2. *Identify parts of code an external library affects.*
3. *Investigate specific contexts.*
4. *Support goal-directed, hypothesis-driven comprehension.*
5. *Highlight the flow of inputs to output through a script.*
6. *Understand syntactic and semantic relationships between variables and functions.*

3.2 Language-Level Provenance

Researchers have collected provenance at various layers and granularities. We define three levels of digital provenance: system, application, and language. CamFlow is a tool that gathers provenance at a whole-system level. This granularity includes various kernel related entities and activities, including inodes, messages, network packets, tasks, users, and groups [26]. Researchers use this level for system intrusion and fault detection. Application-level tools, such as VisTrails, assist in workflow management [7]. They collect provenance on parameters, inputs, and outputs to software over multiple iterations. This information helps users who are in an exploratory phase of their work to keep track of their progress. While we explored the design space of these two levels, our primary focus is on language-level provenance (LL-Prov).

Language-level provenance (LL-Prov) is a historical record of the execution of a program at the source code level. LL-Prov contains information at a line-by-line scale for control flow, values, types of variables, libraries imported, and their relationships. Two tools exist to collect provenance at a language level: *noWorkflow* [24] and *RDataTracker* (RDT) [18]. These tools collect provenance for Python and R, respectively. They are intended to be used in data analysis. While users can write programs in Python and R that do more than simply data analysis and produce results, provenance of other execution is outside the scope for these projects. There are also differences between the two tools' motivation and background. NoWorkflow does not collect LL-Prov by default; instead, it tracks and visualizes application-level provenance. Collecting LL-Prov requires a user to specify they want a finer grain collection. It stores its provenance data in a local SQLlite database with a schema defined by the authors.

In contrast, RDT is a provenance collection tool for R whose goal is to increase scientific reproducibility. Provenance can help verify a

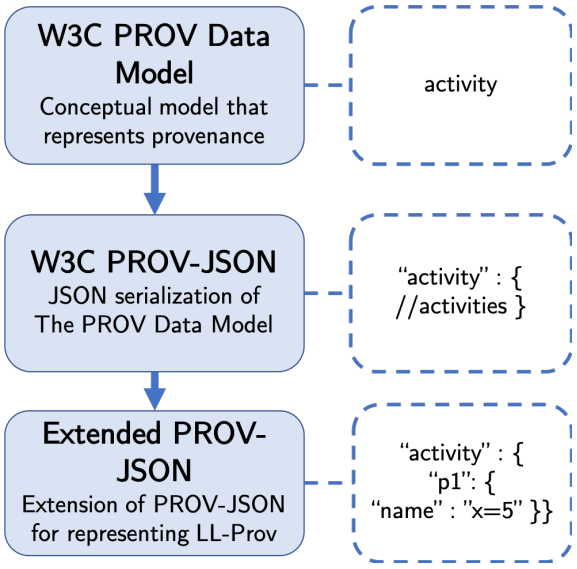

Fig. 3. The PROV Data Model and its implementations. As an example, the PROV model defines the idea of activities. PROV-JSON is simply implementing the concept of activities in JSON format. Extended PROV-JSON states that specific activities needing representation are procedures (denoted with the tag p). Procedures correspond to units of code, such as a single line that is executed. A full Extended PROV-JSON example is in appendix A.

script's result by showing how the script came to its outcome as well as its computing environment. This tool and the output it produces is the focus of our project. It collects LL-Prov in the form of a directed acyclic graph (DAG) on a per-execution basis. While it is possible to use this tool for application-level purposes like noWorkflow, RDT does not contain this as a feature. Instead, users would have to create their own workflow management system that uses its data. RDT has two implementations as R packages, *rdt* and its subset, *rdtLite*. We choose to utilize the rdtLite flavour of RDT as the rdt R package contains additional features, tools, and dependencies that are not necessary to purely collect LL-Prov. RDT collects LL-Prov network data in an extended format of the W3C PROV-JSON model [13]. PROV-JSON [15] is a serialization of the PROV Data Model defined by the W3C [4]. This relationship is shown in Fig. 3.

The nodes in Extended PROV-JSON represent lines of codes executed, data (such as files and variables), external libraries used, and functions used from these external libraries. The edges represent control flow, data flow, and connections to the external libraries. An important fact to note is that a PROV-JSON file represents a *single* execution of a script. A subsequent execution is sure to produce a different provenance graph; at a minimum the timestamps differ. However, a possibility is that numerical results can vary, even with the same input data. This change can be a product of the operating system or versions of software used [11].

The provenance graph consists of seven types of nodes and five types of edges that form a DAG. We describe this data model here, but real examples of LL-Prov in JSON format can be found in appendix A.

The Nodes

In a real-world data analysis the number of nodes is typically in the low 100s. There are multiple types of nodes as discussed here.

Procedure nodes represent a unit of code the script executed. Most commonly, a unit of code is a single line that was executed at runtime. These nodes contain information such as the code executed, how long it took, and where the code is located in the script. In an average script, the amount of procedure nodes will be slightly larger than the number of lines of code. Therefore, the number of procedures vary as much as the number of lines of code in a script.

Data nodes represent information stored in the script, typically variables and files. They store the value (if possible), type of the value, type of the data, and scope. In a script, the number of data nodes will vary with the number of times a variable receives a value and a file is read in or out.

Library nodes correspond to a package loaded into the R environment for use in the script and its version. There are usually not many of these nodes, as there will be one for every library imported for the script; however, as certain packages are always included like the *base* package.

Function nodes represent a function called in the script that was not defined in the script, and therefore was taken from a package. The number of these types of nodes will vary with how extensively the user relies on calling functions from libraries.

The environment node holds information about where and when the script was executed. This includes the architecture and operating system of the computer, the path to the file, paths to relevant directories, and how long it took to run. There will only ever be one of these nodes per JSON file.

The agent node stores information about the versions and settings of provenance being collected. There will only ever be one of these nodes per JSON file.

The prefix node contains information about where the formatting of various parts of the JSON come from. Nodes and edges will contain a prefix that indicate whether the JSON is formatted to as part of the PROV-JSON model, or the extension of PROV-JSON defined by RDT. Therefore, the two prefixes will either be *prov* or *rdt*. There will only ever be one of these nodes per JSON file.

The Edges

In real-world analyses, there are typically from the low to mid 100s of edges.

Procedure to Procedure edges represent control flow. Essentially, in the pair of procedure nodes p1 and p2, this edge means after p1 executed, the next thing to happen in the script was p2. The amount of these edges will always be equal to the number of procedure nodes minus one.

Procedure to Data edges link data nodes to where they were first initialized or brought into the script. The amount of procedure to data edges should always be less than or equal to the number of data nodes.

Data to Procedure edges correspond to where data nodes were used after their creation. The number of these will vary greatly depending on the script.

Function to Procedure edges link function nodes to where they were used in the script. The amount of these edges will depend on how often functions are called from external libraries, which varies greatly.

Library to Function edges link the R package a function node came from. The amount of these edges will depend on how often functions are called from external libraries, which varies greatly.

The RDT authors defined this specification derived from the PROV Data Model as the abstraction for source code at the level of information they are gathering. However, L-Vis and its users do not need this level of specificity for the purposes of their tasks. We define our own abstraction from the provenance domain based on these tasks. We do not need every type of node in our abstraction as some nodes are intended as metadata. Specifically, these are the prefix, agent, or environment node.

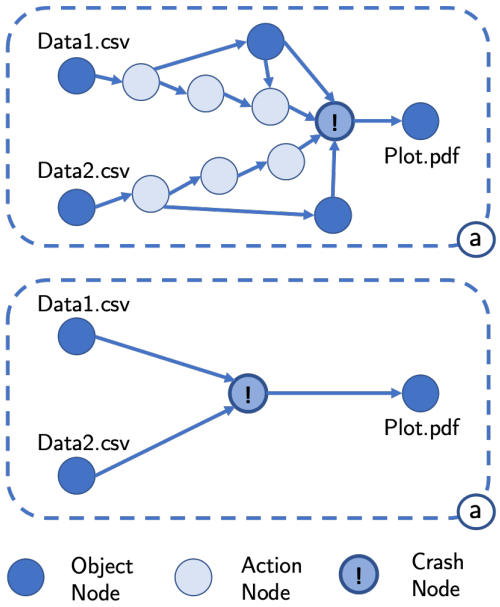

Fig. 4. To show how we filter data, the node-link diagram in (a) displays a provenance graph where transformations are applied to two datasets before they are combined to produce a plot. Our filtering method is shown in (b) with the same analysis, except only the necessary data and crash nodes are visualized.

L-Vis Data Abstraction

Actions are simply procedure nodes defined by the Extended PROV-JSON. For the user, this data will commonly be a single line of code executed in a script.

Objects are any sort of data that is used in an analysis. This can represent variables, files, libraries, and external functions. Objects can be created and used by actions.

Crash Nodes are actions nodes that use two (or more) objects and generates a new object. This is a derived abstraction used to filter the data to highlight the flow and interaction of data through a script. Filtering by crash nodes are visualized in Fig. 4.

Relationships are any of the edges defined in the Extended PROV-JSON. These connections indicate object and action generation, usage, or chronology.

4 Solution

Based on our review of prior work and informal qualitative interviews, we developed a new browser-based visualization tool called L-Vis. Motivated by our task requirements and issue of scale in node-link graphs, we consider two solutions: the traditional provenance approach of filtering a subset of "important" nodes, and a visual idiom with a semantic layout algorithm: the hive plot [17].

4.1 Data Filtering

To reduce the scale of the visualized dataset, we choose to create an option to filter nodes and explicitly show *crash nodes*. Crash nodes are actions where two (or more) different objects are both used and a new object is created. This process is shown in Fig. 4. By filtering all the actions to just crash nodes, the focus of the visualization becomes the path of data through the scripts, and how different inputs may interact and depend on each other.

It is important to track how an analyst transforms a single dataset through a script; however, when multiple inputs come together to create an output, the dependencies become more complicated and could benefit from visual assistance. Choosing to visualize crash nodes reduces scale of provenance data: from the scale of 1000s to 10s or 100s. Based on our task requirements, we understand that users value being able to view the entire dataset. Therefore we allow users to toggle between a filter of the crash nodes and the full dataset, where the full dataset leverages a separate graph layout algorithm. We acknowledge that users may find traditional layout algorithms more accessible based on prior work [5], but discourage users from this view as the network structure will likely regress to a hairball [23] or be too long to fully display.

4.2 Semantic Graph Layout: Hive Plots

An issue with graph layouts is that any inferred structures are by-products of the layout algorithm rather than the underlying structure of the data itself. Hive plots were designed to use a semantic layout algorithm where all nodes are grouped onto the same axes according to some related category [17]. Hive plots therefore encode information into their structure, improving their expressiveness when compared to traditional graph layout algorithms.

Hive plots group nodes onto an axis by category, such as in-node, out-node and procedure node, by time of creation, or even by derivation. Explicit encoding of position by category can allow readers to better infer structure of the data while also communicating additional information. L-Vis orders nodes along an axis by link degree; nodes towards the center of the axes would be less connected (indicating they affect fewer nodes), while nodes at the end of the axes would have higher degrees of connectedness and affect many sections of the codebase. This spatial encoding can give edges less room for occlusion.

A design choice necessary for hive plots is to choose how nodes are grouped along axes. While this choice can be made somewhat arbitrarily, we use our tasks to inform our design. Users noted that they wanted to track inputs and outputs of various objects through the script. Therefore in L-Vis we group nodes along axes categorized by inputs and outputs. The intermediary axes denote intermediary inputs and intermediary outputs. These are reflected into two axes, as many inputs during the execution of a script may feed into outputs and vice-versa. Another design decision regarding the axes is the angle orientation. For L-Vis, we place the inputs on the axis pointing in the 12:00 direction, and the outputs in the axis pointing in the 8:00 position. The axes pointing towards 4:00 and 5:00 represent the data flow through the script with the nodes mirrored across the axes. This allows users to still read the chart somewhat from left to right in a clockwise manner, and focuses user attention on the intermediary input/output axes.

We encode actions as yellow points and objects as blue points on the various axes. Any relationships between nodes we encode as light gray lines that curve towards their destination; however, if a user hovers over a node or edge, these connections will darken as seen in Fig. 1. In addition to bringing attention to connection, clicking a node will display its information in the upper left corner of the main canvas. On the right edge, L-Vis has options to filter by various types of relationships between objects and actions.

4.3 Interface Design

We split the interface of L-Vis into three parts. The script display is on the far left, the main canvas displaying the hive plot of the provenance is in the middle, and optionally the PROV-JSON is on the right. We display this layout in Fig. 5. As reading a page is commonly completed from left to right, we choose to position the script editor far to the left, giving the user context about the program they are trying to understand first. The visualization is the main view, taking about 70% of the view width. We provide a toggle to display the raw PROV-JSON collected from the execution of the script on the far right side. While the intent of L-Vis is to remove the need to view the PROV-JSON, we include this view in case users desire the additional context the raw data provides. In the upper right corner of the main canvas is a legend describing the axes.

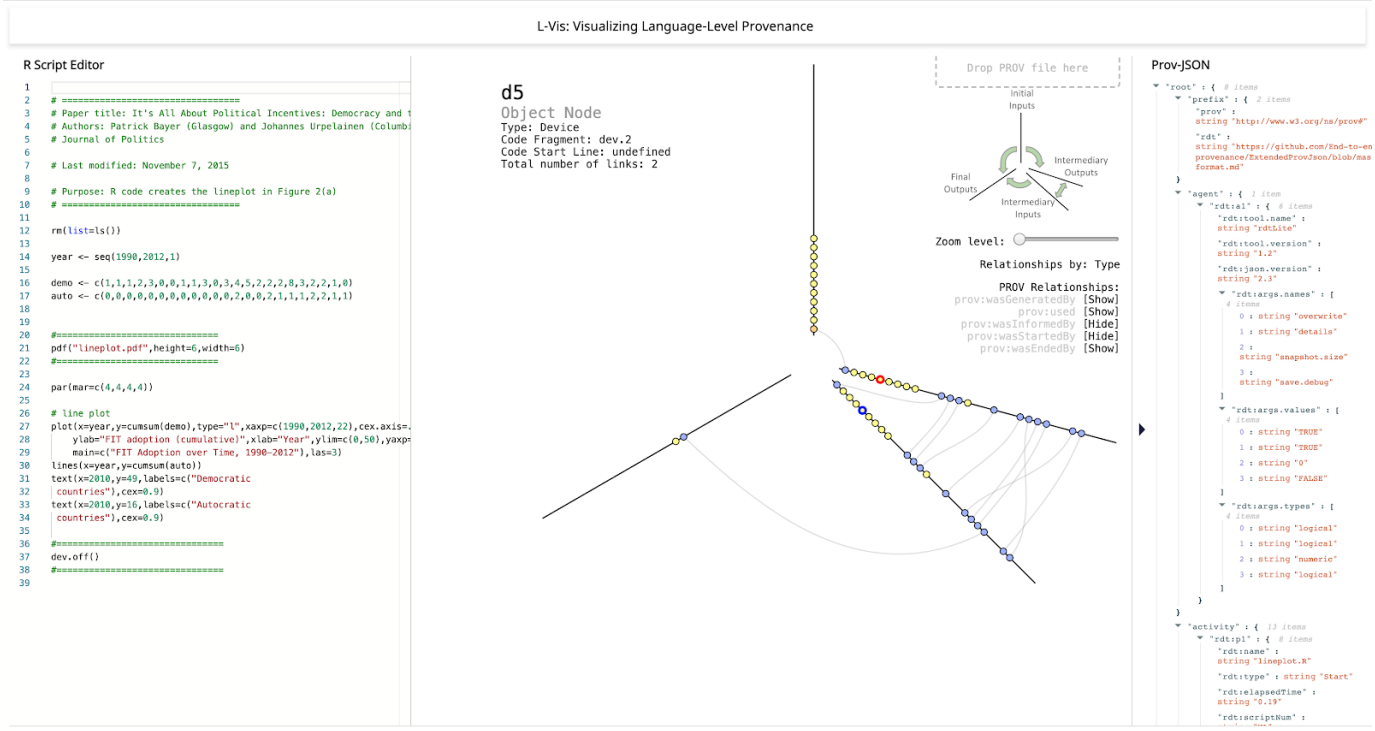

Fig. 5. The L-Vis interface is split into three parts: the script editor on the left, the main canvas with hive plot in the middle, and the PROV-JSON on the right. In the hive plot, the yellow nodes represent actions. The blue nodes represent objects.

5 Implementation

There were two main elements to the implementation of L-Vis: the data collection platform, containR, and the L-Vis interface itself.

5.1 Data Collection Platform

We integrated L-Vis into a prior R analysis reproducibility tool: containR[2] [8]. ContainR is a service provided as a website that leverages RDT, Docker [21], and Dataverse [16] to help archive analyses and increase their repeatability. Fig. 6 depicts the architecture of containR. Users can upload analyses to containR, and they have the option to pre-process the scripts for any common errors that can occur with these R analyses. These errors could be working directory issues or uninstalled libraries. This method is by no means a guaranteed fix for broken analyses, but may just unmask other problems as found by the original author of containR [8]. If rdtLite can successfully execute each R file in the uploaded directory, Docker bundles the scripts, data, results, and provenance into an RStudio Server container and uploads it to Docker Hub. When the user wants to repeat their experiments, they can pull the container, start up the server, and execute the analysis. Since the Docker Hub repository is public, anyone who wants to repeat the analysis and has Docker installed can pull the container and run it, even if they do not have R installed locally.

We wanted to use this workflow to collect provenance of analysis scripts on Dataverse which we would use as input to L-Vis. However, the site was never put into production. We inherited the containR base, a Python Flask application, which unfortunately did not run to to a variety of issues. These include deprecated dependencies, database schema changes, lack of database at all and deprecated provenance collection tool. To fix these bugs, we went through the tedious process of updating Python package requirements, creating a local SQLite database to connect to and updating the database schema. We could not find all dependencies based on stated requirements, and instead we found them through trial-and-error. The other significant change to the site that we made was the provenance collection tool. When we inherited the code it was using a deprecated LL-Prov collection tool. We removed it and switched in rdtLite, which its authors are currently supporting and updating. To provide L-Vis the most information possible, we ensured that not only was the PROV-JSON being placed into the container, but the entire provenance directory that rdtLite creates upon execution. This directory includes the PROV-JSON as well as the original script.

[2]This section of the project was done for the purposes of CPSC 508. Had this been for 547 alone, we would have made L-Vis a standalone tool.

5.2 L-Vis Interface

The L-Vis interface was implemented in Javascript using the following libraries: D3, React, Parcel, Monaco Editor and React JSON viewer. We used react-split-panel to layout the page although style the rest of the page ourselves.

One main development effort was the data pipeline. For the data pipeline, we needed to transform PROV-JSON collected by RDT into a source-target node and link data format usable for hive plots in D3. There were no previous examples of data transformation from PROV-JSON to D3 readable source/destination that we could build on. Therefore, we wrote a Python parser (convert.py) that would parse any loaded PROV-JSON and transform the data into the source/destination node format that D3 can use.

Another challenge was implementing hive plots in D3. There are previous examples of hive plots. However, the majority of past examples either (a) used mock data where the information encoded was not complex or (b) ordered nodes along the axes in a simple way. While there is a D3 hive plot submodule, this module only contains the layout algorithm and shape generation code for the links of a hive plot. Additionally, there were compatibility issues between the code found in previous examples. All hive plots examples used D3.v2 while we used D3.v5. We based the skeleton of our hive plot implementation on Mike Bostock's example on bl.ocks.org [6], trying to change any bugs that arose from deprecated APIs in the transition to D3.v5. In order to determine the layout of the nodes, we computed a rank for each node based on its in-degree and out-degree and by node type, then interpolate

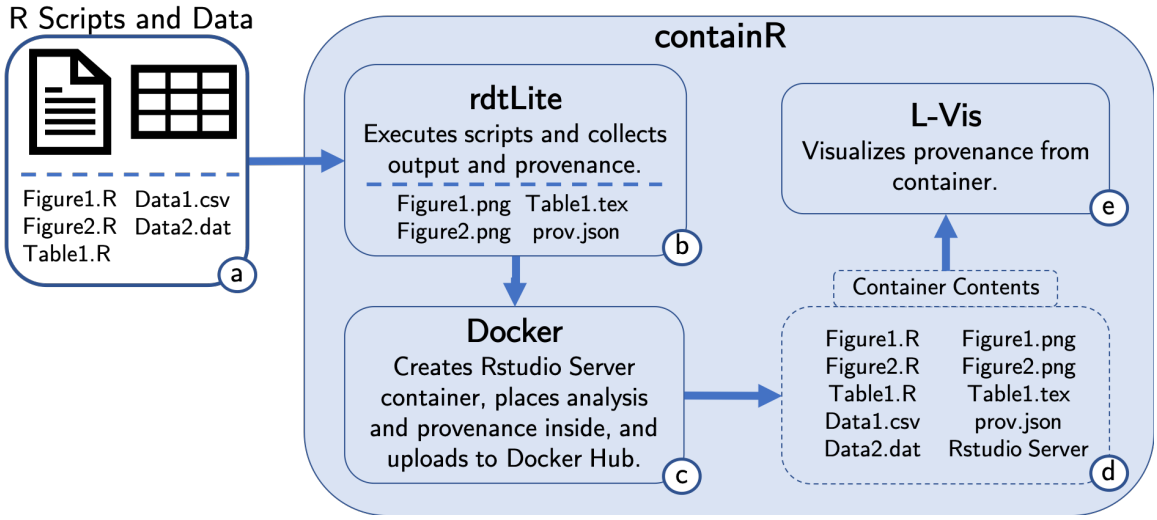

Fig. 6. Dependencies and data flow within containR. Users upload analyses and data to containR (a). ContainR runs the rdtLite package to execute scripts, store their outputs, and collect provenance (b). Upon completion, ContainR runs Docker to containerize all the scripts and data and uploads the new container to Docker Hub where it is publicly accessible (c). Uploaded container links are displayed to users on the containR homepage where they can visualize container contents (d) using L-Vis (e).

across the axis length according to this rank. Then, we simply allow the D3 hive plot submodule to generate the edges between the nodes.

6 Results

L-Vis can successfully help users accomplish tasks we defined for the tool. The hive plot idiom helps address the long linear layout that is difficult for users to parse introduced by the traditional node-link graph. The axes are defined for best showing dataflow through an analysis. The hive plot can still encounter the hairball problem prevalent with network visualizations; however, our crash nodes data filter can help control this problem for certain scripts. With all this information, L-Vis can help users accomplish specific tasks from our requirements.

6.1 Success Scenario

A user can use L-Vis to assist them in trying to switch a deprecated R package out for a new one. They can open the tool, find the object corresponding to the library they need to switch out, and click it. As shown in Fig. 1, this will highlight its connections to the objects, functions in this case, that are from this library. They can then use the relationships to find the actions that use those objects. Since each of those actions correspond to a line of code, they now have all the places in their code where that library was used. If they are concerned about functions returning specific types of objects, they can even trace the dataflow from the connected actions to find everywhere they may have to account for a new type of object from the new library. This same system can be used to track how inputs become outputs, another defined task. Rather than clicking on a library object to start, a user can click an input object. While all these features are shown in L-Vis, there are still some we want to integrate.

6.2 Informal Qualitative Feedback

In an ideal world with additional resources and time, we would run a formal quantitative user study where we compare L-Vis to alternatives, like DDG Explorer.[3] However given the time constraints and scope of the course, we asked for feedback from our CS graduate peers who kindly volunteered their time to use and critique L-Vis. We asked participants to be somewhat familiar with the R programming language as minimal background knowledge for the study. All participants used the same Macbook Pro 13" laptop running macOS Mojave and track pad to navigate DDG Explorer and L-Vis.

Participants were given a scenario where a package dependency had changed and they would have to replace the package. The dataset used was scraped from Harvard Dataverse [16], a archival service for scientific analyses, and was a simple script drawing a line plot for the result of a scientific analysis.

During the feedback, participants are asked to speak-aloud their process so that the observers would be able to better understand how participants were developing their cognitive models of the analysis code. Having participants verbalize their process was important for allowing us to assess participant usage of L-Vis'.

We asked 3 of our peers to use L-Vis. They provided feedback that will be used in the next iteration of L-Vis, such as the addition of linking highlighting between the visualization and code snippets. For example, a peer noted that, "it helped a bit, but more functionality has to be built before I would consider using it for something useful." Ultimately, this cannot take the place of a formal user study, however has helped begin an iterative design process to eventually converge on a better solution that might include L-Vis.

6.3 Limitations

L-Vis has some drawbacks from lack of implemented features as well as the nature of the data and tasks. There are a few features that we would have implemented given more time. For example, in the previous scenario with replacing a deprecated library the user could benefit from bi-directional highlighting. When a user clicks on a library object, it would be useful to have a feature that would trace all the relationships previously described and map them to the corresponding line numbers in the script editor. These lines would then be color encoded to highlight them as the relevant code blocks. Since this system would be bidirectional, the same would go in reverse. A user could click on a line and L-Vis would highlight all the related nodes and edges. Additionally, we could expand and update the label system. Currently the nodes still display the PROV-JSON prefixes, *prov* and *rdt* before each node. This is metadata not useful to the user when they're trying to comprehend a program and we could filter them out.

7 Discussion and Future Work

Taken as a whole, the results from our user study show that L-Vis' new visual idiom and additional tools led to more accurate tasks completion

[3] This qualitative evaluation was a component required for CPSC 508. We add this section to this paper for completeness.

and higher qualitative results than alternatives such as DDG Explorer. The hive plot, while initially hard to learn, encoded semantic information meaningfully for users through position, and thus was preferable to the traditional node-link graph diagram represented in DDG Explorer. The qualitative results supported this, showing a significant effect size of preference to L-Vis over DDG Explorer.

Through the use of the hive plot visual idiom, L-Vis is less susceptible to the scalability hairball problem encountered when visualizing large graphs. DDG Explorer encounters scalability issues, although not always due to a hairball. Long scripts have a linear structure due to the nature of the DAG. DDG Explore will then only be able to visualize small sections of the graph legibly. L-Vis can handle the long linear structure of the graph, but can still encounter the hairball at hundreds of nodes. However a user filtering by crash nodes can reduce the problem while still preserving the structure of data flow. In addition to scalability, L-Vis helps keep the users oriented with their script by displaying it in a window next to the visualization canvas. The platform of L-Vis also allows it to be accessible as long as someone has a modern web browser.

We acknowledge that the tasks conducted in the study might not be representative of common tasks in scientific analysis. In particular, this would be strengthened by conducting an initial pilot assessing a representative set of tasks that demonstrate programmers true workflows when attempting to interpret an unfamiliar script. In addition, the targeted user group and participant pool was slim, reducing the generalizability of our results.

Finally, while we tried to counter learning effects, it is possible that prior experience with analysis, provenance, or reproducibility could have confounded the results of our study. A large sample size would certainly help counter-balance these effects and we strive to complete a larger user study in the future, directly evaluating and reproducing the effects found in this study.

7.1 Generalization to Other Provenance

We designed L-Vis specifically for the tasks we defined in our informal interviews. Therefore, the infrastructure of L-Vis might not generalize to other forms of provenance immediately. For example, the axes of the hive plot are defined in a way that makes sense for a provenance graph representing an R analysis script. The plot starts with data inputs, and end with data outputs. We derive crash nodes as a data abstraction and way of filtering nodes while maintaining data flow. Crash nodes may not hold the same value for system or application level provenance. However, with effort, the L-Vis concept could be implemented for other forms.

The hive plot idiom could be used for other scales. To address issues like the hairball, new ways of aggregating or filtering may be necessary. For example, the time-based hierarchy defined for use in InProv could be applied to L-Vis. However even with additional filters or aggregations applied, the axes and what they represent will need to be defined for these other levels.

7.2 CPSC 508 vs 547

If L-Vis were a project just for CPSC 547, we would not specify containR as the platform for our visualization. Rather, we would have created a standalone tool that uses Extended PROV-JSON as its input. We also would have focused on comparisons between additional visual idioms, like traditional node-link graph layouts or radial layouts. Specifically, we would have implemented these other idioms within L-Vis and let users toggle between them in the interface.

If L-Vis was a solo project for CPSC 508, we would describe less work of the domain/task abstraction and more in the work to update containR and ensure compatibility with supported provenance collection tools. Extending containR to add L-Vis was necessary to add a more systems-like component to the project. In this paper we refer to various nodes in their abstracted form: actions, objects, relationships, and crashes. If just in CPSC 508, we would describe these in the domain-specific setting.

It is important to note that regardless of the class, we still would have conducted unstructured interviews with stakeholders. This was a key step in understanding the requirements of our target users and defining tasks (domain specific or not) that L-Vis should help support. When corroborated with prior work in program comprehension, this led to the requirements list which could be further task-abstracted for 547.

8 Conclusions

We define a set of domain and task abstractions for language-level provenance and present L-Vis as an alternative tool to visualize LL-Prov in a comprehensible form. We find that users qualitatively prefer L-Vis to DDG Explorer and while users expressed initial confusion interpreting the hive plots in L-Vis, believe hive plots to be more visually effective than their alternatives. While not directly generalizable to other levels of provenance, L-Vis has the potential to be applied to other scales with some consideration made of data and task abstractions.

L-Vis is still in development; we are constantly adding tools to improve the feature set. Based on the feedback and results from our user study, we aim to continue development on L-Vis where it can be integrated into containR. In particular, we target how to make L-Vis more accessible in users' current workflows for analysis and reproducibility. We understand that users desire as little intrusion into their current workflows as possible and seek to make L-Vis as painless as possible to use.

Furthermore, we plan to incorporate multiple graph layouts in L-Vis for LL-Prov. By implementing multiple idioms, we can directly compare L-Vis' hive plots to other visual representations such as a radial layout for LL-Prov adapted from InProv [5]. The design space of alternative visual encodings for provenance information is slim and comparisons between novel visualization types have not been explored and evaluated yet. We believe future work in this domain has promise to determine the most effective visual idiom for LL-Prov. Finally, while we opted to filter LL-Prov to visualize crash-nodes, we plan to test L-Vis on larger dataset sizes where crash-nodes exceed the number of a few hundred.

We hope that providing an intuitive summary of code-structure in scientific analyses with L-Vis allows users to more effectively generate cognitive models of code they are unfamiliar with. Our goal is to align these mental models with those of the original author. We also hope that students who inherit spaghetti "grad-student code" will be able to make sense of it through L-Vis.

References

[1] J. Abello, F. van Ham, and N. Krishnan. ASK-GraphView: A large scale graph visualization system. *IEEE Trans. Visualization and Computer Graphics*, 12(5):669–676, September 2006. doi: 10.1109/TVCG.2006.120

[2] D. Archambault, T. Munzner, and D. Auber. Grouse: Feature-Based, Steerable Graph Hierarchy Exploration. In K. Museth, T. Moeller, and A. Ynnerman, eds., *Eurographics / IEEE-VGTC Symp. on Visualization*, 2007. doi: 10.2312/VisSym/EuroVis07/067-074

[3] D. Archambault, T. Munzner, and D. Auber. GrouseFlocks: Steerable exploration of graph hierarchy space. *IEEE Trans. Visualization and Computer Graphics*, 14:900–13, 07 2008. doi: 10.1109/TVCG.2008.34

[4] K. Belhajjame, R. B'Far, J. Cheney, S. Coppens, S. Cresswell, Y. Gill, P. Groth, G. Klyne, T. Lebo, J. McCusker, and et al. PROV-DM: The PROV Data Model, 2013.

[5] M. A. Borkin, C. S. Yeh, M. Boyd, P. Macko, K. Z. Gajos, M. Seltzer, and H. Pfister. Evaluation of filesystem provenance visualization tools. *IEEE Trans. Visualization and Computer Graphics*, 19(12):2476–2485, December 2013. doi: 10.1109/TVCG.2013.155

[6] M. Bostock. Hive plot (links). Published on Mike Bostock's blo.ocks.org account., 2019.

[7] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. VisTrails: Visualization meets data management. In *Proc. ACM SIGMOD International Conf. Management of Data*, pp. 745–747, 2006. doi: 10.1145/1142473.1142574

[8] C. Chen. Coding better: Assessing and improving the reproducibility of r-based research with containr. Undergraduate thesis, Harvard University, 2018.

[9] B. Cornelissen, A. Zaidman, A. van Deursen, and B. V. Rompaey. Trace visualization for program comprehension: A controlled experiment. *IEEE International Conf. Program Comprehension*, pp. 100–109, 2009.

[10] H. A. Duru, M. P. Çakir, and V. Isler. How does software visualization contribute to software comprehension? a grounded theory approach. *International Journal Human Computer Interaction*, 29:743–763, 2013.

[11] A. M. Ellison. Repeatability and transparency in ecological research. *Ecology*, 91(9):2536–2539, 2010. doi: 10.1890/09-0032.1

[12] K. Erdos and H. M. Sneed. Partial comprehension of complex programs (enough to perform maintenance). In *Proc. 6th International Workshop Program Comprehension*, pp. 98–105, June 1998. doi: 10.1109/WPC.1998.693322

[13] E. Fong. Extended prov-json. RDataTracker GitHub PROV-JSON Specification, August 2019.

[14] E. R. Gansner, Y. Koren, and S. C. North. Topological fisheye views for visualizing large graphs. *IEEE Trans. Visualization and Computer Graphics*, 11(4):457–468, July 2005. doi: 10.1109/TVCG.2005.66

[15] T. D. Huynh, M. O. Jewell, A. S. Keshavarz, D. T. Michaelides, H. Yang, and L. Moreau. The PROV-JSON serialization, 2013.

[16] G. King. An introduction to the dataverse network as an infrastructure for data sharing. *Sociological Methods & Research*, 36(2):173–199, 2007. doi: 10.1177/0049124107306660

[17] M. Krzywinski, I. Birol, S. J. Jones, and M. A. Marra. Hive plots—rational approach to visualizing networks. *Briefings in Bioinformatics*, 13(5):627–644, 12 2011. doi: 10.1093/bib/bbr069

[18] B. Lerner and E. Boose. RDataTracker: Collecting provenance in an interactive scripting environment. In *USENIX Workshop on the Theory and Practice of Provenance (TaPP)*, 2014. doi: 10.1007/978-3-319-16462-5_36

[19] W. J. Longabaugh. Combing the hairball with BioFabric: a new approach for visualization of large networks. *BMC Bioinformatics*, 13(1):275, 2012. doi: 10.1186/1471-2105-13-275

[20] P. Macko and M. Seltzer. Provenance map orbiter: Interactive exploration of large provenance graphs). In *Proc. USENIX Workshop Theory and Practice of Provenance (TaPP)*, 2011.

[21] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239), Mar. 2014.

[22] S. Mohseni, A. M. Pena, and E. D. Ragan. Provthreads: Analytic provenance visualization and segmentation. *ArXiv*, abs/1801.05469, 2018.

[23] T. Munzner. *Visualization Analysis and Design*. CRC Press Taylor & Francis Group, 2014.

[24] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noWorkflow: Capturing and analyzing provenance of scripts. In B. Ludäscher and B. Plale, eds., *Provenance and Annotation of Data and Processes*, pp. 71–83. Springer International Publishing, 2015.

[25] M. J. Pacione, M. Roper, and M. Wood. A novel software visualisation model to support software comprehension. *11th Working Conf. Reverse Engineering*, pp. 70–79, 2004.

[26] T. F. J. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. M. Eyers, J. Bacon, and M. Seltzer. Runtime analysis of whole-system provenance. *Proc. ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[27] Rui Yin and R. K. Keller. Program comprehension by visualization in contexts. In *Proc. International Conf. Software Maintenance*, pp. 332–341, October 2002. doi: 10.1109/ICSM.2002.1167789

[28] J. Sillito and G. C. Murphy. Questions programmers ask during software evolution tasks. In *Proc. 14th ACM SIGSOFT International Symp. Foundations of Software Engineering*, pp. 23–34, 2006. doi: 10.1145/1181775.1181779

[29] M.-A. Storey. Theories, methods and tools in program comprehension: past, present and future. In *International Workshop Program Comprehension (IWPC)*, pp. 181–191, May 2005. doi: 10.1109/WPC.2005.38

Appendices

A LL-Prov JSON Format

A.1 Procedure Node

```
{"rdt:p2": {
        "rdt:name": "my.func <- function(x) {x * x + 5}",
        "rdt:type": "Operation",
        "rdt:elapsedTime": "0.132",
        "rdt:scriptNum": 1,
        "rdt:startLine": 1,
        "rdt:startCol": 1,
        "rdt:endLine": 1,
        "rdt:endCol": 33
}}
```

A.2 Data Node

```
{"rdt:d2": {
        "rdt:name": "foo",
        "rdt:value": "10",
        "rdt:valType": "{\"container\":\" vector\", \"dimension\":[1], \" type\":[\"numeric\"]}",
        "rdt:type": "Data",
        "rdt:scope": "R_GlobalEnv",
        "rdt:fromEnv": false,
        "rdt:hash": "",
        "rdt:timestamp": "",
        "rdt:location": ""
}}
```

A.3 Library Node

```
{"rdt:l1": {
        "name": "base",
        "version": "3.6.1",
        "prov:type": {
                "$": "prov:Collection",
                "type": "xsd:QName"}
}}
```

A.4 Function Node

```
{"rdt:f1": {
        "name": "read.csv"
}}
```

A.5 Environment Node

```
{"rdt:environment": {
        "rdt:name": "environment",
        "rdt:architecture": "x86_64",
        "rdt:operatingSystem": "linux-gnu",
        "rdt:language": "R",
        "rdt:langVersion": "R version 3.6.1 (2019-07-05)",
        "rdt:script": "/home/User/Documents/ProvData/example.R",
        "rdt:scriptTimeStamp": "2019-12-06T14.02.07PST",
        "rdt:totalElapsedTime": "1.164",
        "rdt:sourcedScripts": "",
        "rdt:sourcedScriptTimeStamps": "",
        "rdt:workingDirectory": "/home/User/Documents/ProvData",
        "rdt:provDirectory": "/home/User/Documents/ ProvData/prov_example",
        "rdt:provTimestamp": "2019-12-06T14.02.35PST",
        "rdt:hashAlgorithm": "md5"
}}
```

A.6 Agent Node

```
{"agent" : {
        "rdt:a1": {
                "rdt:tool.name": "rdtLite",
                "rdt:tool.version": "1.1.1",
                "rdt:json.version": "2.3",
                "rdt:args.names": [
                        "overwrite",
                        "details",
                        "snapshot.size",
                        "save.debug"
                ],
                "rdt:args.values": [
                        "TRUE",
                        "TRUE",
                        "10",
                        "FALSE"
                ],
                "rdt:args.types": [
                        "logical",
                        "logical",
                        "numeric",
                        "logical"
                ]
        }
}}
```

A.7 Prefix Node

```
{"prefix": {
        "prov": "http://www.w3.org/ns/prov#",
        "rdt": "https://github.com/ End-to-end-provenance/ ExtendedProvJson/blob/master/ JSON-format.md"
}}
```

A.8 Procedure to Procedure Edges

```
{"rdt:pp1": {
    "prov:informant": "rdt:p1",
    "prov:informed": "rdt:p2"
}}
```

A.9 Procedure to Data Edges

```
{"rdt:pd1": {
        "prov:activity": "rdt:p2",
        "prov:entity": "rdt:d1"
}}
```

A.10 Data to Procedure Edges

```
{"rdt:dp1": {
        "prov:entity": "rdt:d1",
        "prov:activity": "rdt:p5"
}}
```

A.11 Function to Procedure Edges

```
{"rdt:fp1": {
        "prov:entity": "rdt:f1",
        "prov:activity": "rdt:p5"
}}
```

A.12 Library to Function Edges

```
{"rdt:m1": {
        "prov:collection": "rdt:l9",
        "prov:entity": "rdt:f2"
}}
```