CSCI 578 Final Project Report

Ryan Chase, Negar Abolhassani, Lucas Rebelo, Vahagen Sinanian

1.Introduction

Current visualizations for recovery methods are overpopulated with information. To make them readable, lots of information must be removed which can be seen in both RELAX and ARC visualizations. In ARC we only see dependencies while in RELAX we have the system's structure and clusters. However, the output is still huge and missing information, so it is not easy to follow dependencies or to look for specific files. In order to cover all available information in a readable way, we propose an interactive visualization which consists of three parts. We visualize a system from three perspectives, structure, dependencies, and clusters. We implemented these visualizations using D3.js and the overview of our tool is shown at figure 1. Our application consists of three tools to generate each visualization and we discuss them in detail below.

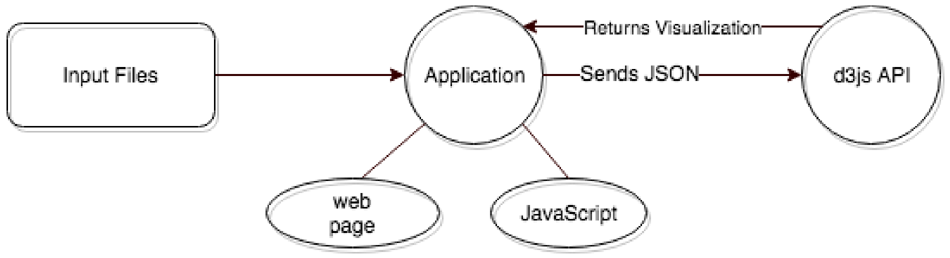


Figure 1

2.Background

Running Arcade on different systems results in several output files. For our application, we need clustering information, dependencies information and source code of each system we want to study. Clustering information can be found in cluster(s/ed).rsf files and deps.rsf files contain dependencies. Our application is written on top of different D3.js libraries to create an interactive environment for each visualization. D3.js accepts JSON files as input, so we need to generate different JSON files with respect to architecture recovery methods' output files. In next section, we describe how we generated the input files for each visualizations and how we implemented each tool.

3.Implementation

To produce each visualization, we designed and implemented a tool. Each tool looks at the system from a different perspective. The reason for this separation is to make our application more flexible and to prevent the application from doing extra work. Users of our system are able to only use a specific part of that based on their needs. For example, a developer interested in reviewing dependencies can use the system and generate results of system dependencies without generating other visualizations.

3.1 Integration

We used Scrollama.js library for integrating all tools in a single page to make it user friendly. We defined three steps for three tools we have. As we scroll, we reach each tool and its specific javascript functions are called to clean the graph area, build the necessary user interface and when clicked generate the graph (except in the clustering portion where a gif of the visualization is shown, with a link to a website running the actual tool. We did that because the visualization is too large to wholly be placed in the page.

3.2 Structure

The goal of this tool is to generate a visualization with respect to the source code. It helps us to dive into an existing project and have a bird's eye view of the whole code. It is useful for architects and developers to get a high-level idea about a system's structure without going through the code.
We developed our visualization on top of a D3.js library named CodeFlower. This library visualizes source repositories using an interactive tree. We adapt the code to make it more flexible and we developed a java code to generate its input file.

3.2.1 Input

The input for this tool is a JSON file containing the information of the source code. To generate a JSON file from a system's code. We used a Java code which is exported as structure.jar. This Jar file go over the directories and files of a system starting from a path given to it from the input. It also uses dependencies.rsf file to find class dependencies. It computes the lines of code in each file. It does not take blank lines into account. It also stores the path of each file or folder to help users understand where to find them in the system's code. The jar file also accepts file extensions to consider those file types and ignore other file types. The general command to run the jar file is:

```
java -jar structure.jar {path to the root directory}[mandatory] {path to the dependencies.rsf file}[mandatory] {system's name}[mandatory] {specific file formats}[optional]
```

The Java code starts by finding class dependencies. Then based on the root path, it recursively creates JSON objects for each file or folder and add all files and folders inside it as its children. For each file, its lines of code are counted and stored in the file JSON object. Folders' size is the sum of line of codes of their children. Class dependencies are used for class files. For each class file, the number of classes dependent on it(dependers) and the number of classes it depends on (dependees) are also stored in the respective JSON file.

The output is a JSON file named {system's name}_struct.json which can be found in the same directory as structure.jar.

The reason for having this part separately and in Java is that Javascript is not able to go through the files and folders locally. Javascript is responsible for frontend and visualization, so generating the input should be done in backend. The Java code used for input generation might be considered a part of ARCADE. Meaning that a user selects the root directory, and specifies file types in ARCADE's user interface and ARCADE generates the appropriate input for the visualization. The input needs to be done anytime that there is a change in the source code not everytime we want to produce a system structure visualization.
The source code of structure.jar is available in structureJavaCode directory.

3.2.2 Graph

After generating the files we go to index.html and the first part is responsible for generating structure visualization. After selecting the input file which was generated before, a user must click the submit button to generate the results. After submitting the input file, you see a graph with a black node located at the center of the graph. The size of the graph is proportional to the total number of nodes of the graph.
Nodes might be either files or folders found in the target system. The Black node always shows the root node. The size of each file node is based on lines of code found in that file. Folders have the same small size all over the graph. Pointing on each node you can gain information about files and folders in the system. All nodes have names and lines of code information. Using dependencies files, a class file on the graph displays number of dependers and the number of dependees.
Users always can click on a node to go through the structure starting from that node. In other words all other nodes will be removed. The code flower code is based on the JSON file and everytime we click on a node it calls the update method with the current JSON object and this process is defined in onClick call back. A user can always go back to the initial state of the graph by clicking the reset button. Reset button onClick callback is defined to update the graph using the initial root JSON object.

3.3 Dependencies

This tool compares package names and creates multiple dependency visualizations at different abstraction levels to hopefully provide a good view of the system's concerns. This was developed entirely in Javascript so the only thing the user needs is a "...deps.rsf" from the output of said system's recovery.

With the input chosen, the software logs all dependencies, for all levels of abstraction. For example, this line from the recovery output gets logged as if it were the following dependencies in different abstraction levels:

Abstraction level	depends edu.berkeley.chukwa_xtrace.graph.CausalGraph org.apache.log4j.Logger

1	depends edu org
2	depends edu.berkeley org.apache
3	depends edu.berkeley.chukwa_xtrace org.apache.log4j
4	depends edu.berkeley.chukwa_xtrace.graph org.apache.log4j.Logger
5	depends edu.berkeley.chukwa_xtrace.graph.CausalGrap org.apache.log4j.Logger

Then, for all levels, it produces the list of package names and the dependencies matrix required by the D3.js APIs. The graph generating code was changed to accept an object instead of a JSON file, to dynamically change text size for a cleaner graph, and show more diverse and vivid colors.

Using the slider that appears right above the graph, the user can change levels of abstraction. This is useful to compare major changes in different versions and to gauge separation of concerns as long as the package naming convention follows some sense of this separation (i.e. group packages according to their functionality).

3.4 Clusters

This tool is responsible for displaying clusters found by different recovery methods. Choosing a file from Load directory, system's name is determined. After that a user can select the target recovery method to generate the visualization. After determining the system, it is no longer needed to choose clustering files of that system. Using the drop down menu, a user can select the target recovery method.

3.4.1 Input

The clustering tool of our application has two main directories Load and JSON. The Load directory contains the cluster(s/ed).rsf files from three recovery methods. After running the project.js file on the command line it gets all the existing files in Load directory. It checks which recovery method the file belongs to and calls generate() method to generate the JSON file.
Generate() method gets the filename, type of the recovery method, and system name as its parameters. Then it opens and read the content of file passed to it. There are three JavaScript

Objects in this function. The first object is the recovery method object which holds the recovery method name and an array of cluster objects. Cluster object holds the name of the cluster and an array that holds objects for children of the cluster. The children object holds the name of the child and a parameter for visualization.
The generate method reads the data from the files and divide the input text into lines and lines are being divided based on the space. In another word, the inputted text is being tokenized. Each token is being placed in its target object and after completing, the objects are being added to their target array. At the end it uses the stringify() method to convert the objects into a string so it can write it into a file. This process happens for all the files in Load directory that have recovery method name and cluster(s/ed) in their name with .rsf file format. The name of the file contains <system name - version>_<recovery method>.rsf. All these files are being stored in JSON directory.

3.4.2 Graph

The graph was developed on top of D3.js. It exploits the JSON file generated from each .rsf file. Cluster nodes are created from the root JSON objects of the JSON file and clicking on each of them results in showing the content of that cluster. It dynamically tries to fit all expanded information on the screen. Using the drop down menu, clusters of the same system from another recovery method perspective can be displayed.

4.Discussion

Usage of these tools provides a more general view of the system leveraging the recovery outputs to provide:

- A bird's eye view of the whole code, showing the structure of the system that allows the architect or developer to gauge the system's complexity without looking at the code.
- An interactive way to gauge how separation of concerns is implemented in the system, as well as the complexity of its interdependencies.
- Comparison between the different recovery methods clustering algorithm, and how well they truly represent the system.

4.1 Structure

In general, understanding the structure of a system is time consuming and labor intensive. As the system gets larger, this process can be get harder. The structure visualization we proposed, provides a high level view of the system. The Java code developed for input generation provides different options to narrow down the number of files by looking for specific file types or starting from any root directory. This high-level view of a system shows the complexity of the system and it guides developers and architects to prioritize their actions.

4.2 Dependencies

When viewing complete dependency graphs, it is not obvious whether the system's functionalities are properly coupled. Providing a visualization of a system's dependencies in a higher level of abstraction lets the architect or developer check on separation of concerns while ensuring the packages responsible for each functionality (e.g. org.apache.hadoop.mapred signifies the whole MapReduce functionality, and therefore in a higher level of abstraction it is possible to see all is external dependencies).

4.3 Clustering

We believe this visualization is helpful for users of this application because it provides an interactive graph for cluster visualization. It allows the user to expand the graph by clicking on the clusters and discover the children inside that cluster and by clicking on the cluster they can close that branch. This can help the user to have a clear understanding of the cluster name, counts, and children. It also helps users to compare the content of different clusters at a single glance. They are able to switch between different recovery methods using a drop down menu to look at the clusters from different viewpoints.

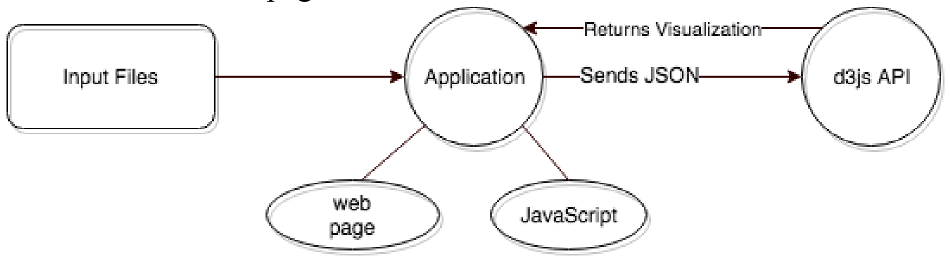
CSCI 578 Final Project Overview

Ryan Chase, Negar Abolhassani, Lucas Rebelo, Vahagen Sinanian

Introduction
We are building a tool for visualization of the recovery method output files.
Our goal is to abstract away some of the details and give the reader a broader, overall view of the system's dependencies, its structure, and how the different recovery methods clustered concerns, all computed by and displayed in a Web App.

Structure of Software

The visualization software that the team plans to build is a web-based application developed in HTML and JavaScript. The application will take the output of the recovery methods as an input, parse the related files and generate a JSON structure for D3js API for visualization. The output of our application will be a web page.



Visualizing Structure

Using directories.dot file in RELAX, we have access to the structure of the system's code. This is independent of the recovery method, so we can use it for all three recovery methods. However, we know that this visualization is feasible if we do not have the RELAX's recovery results, but even without considering that we can go through the source code and find out the structure of the system. After extracting the information about the source code, we are able to create visualizations for it. Using the results from dependencies' data in each recovery method, we are able to detect generators, consumers and critical components in the system structure (we define each component type based on the number of dependencies in and out). This visualization is helpful for developers to prioritize their actions based on the components' types and get familiar with the system's structure at a glance. We also think UCC files (especially TOTAL_outfile.txt) can be helpful to add more information to this visualization, but it needs further investigation.

Visualizing Clusters

To help in the visualization of clusters, we are planning to use an interactive visualization to simplify the graphics produced by the ARC and RELAX recovery methods and also to provide a visualization for ACDC results (using package names as cluster names). *Figure 1* is a prototype of the detailed version of RELAX visualization. We are going to use cluster<s/ed>.rsf files to extract the clustering of each recovery method. Currently, the cluster graphics suffer from being over-populated with information.

We would like to simplify it in the following way:

1. Aggregate arrows in different levels of abstraction by representing with color and size.
    a. Darker colored and thinker arrow means more arrows when we zoom in
2. Position the clusters to be more readable at first glance
    a. Less overlap of arrows and concern overlap
    b. Perhaps imply structure by absolute position
3. Create cluster nodes and label each one with a topic
    a. In RELAX, we label each cluster with its concern's name
    b. In ARC, we label each cluster with the number associated to its topic and clicking on that number result in a the list of words creating that topic

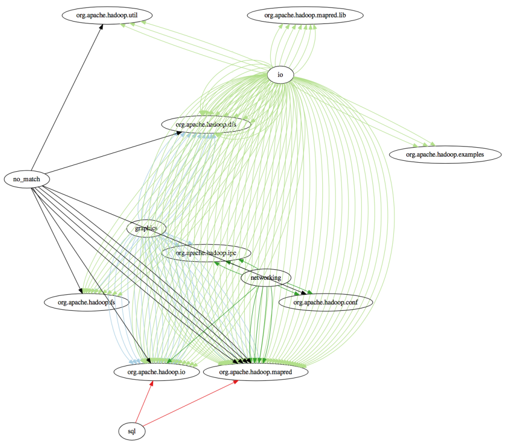


Fig. 1. Clustering visualization

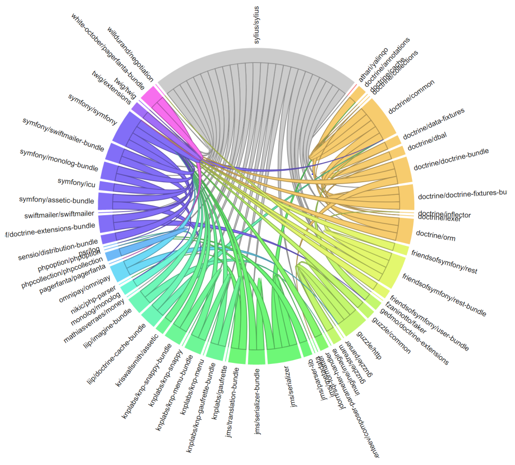


Fig. 2. Dependencies visualization

Visualizing Dependencies

Our goal in providing a new way of visualizing a system's dependencies is to provide a simpler graph representing a higher level of abstraction. This will represent the "big picture" of the system, but also give the ability to seamlessly dive deeper.

Using the deps.rsf file (which is common to all three recovery methods), we plan on remapping the dependencies at the highest level of abstraction (in Javascript), and repeating that for each subsequent lower level of abstraction. For example, from:

- *depends edu.berkeley.chukwa_xtrace.CausalGraph edu.berkeley.xtrace.reporting.Report*

We can extract the following two dependencies, each at different levels of abstraction:

- *depends edu.berkeley.chukwa_xtrace edu.berkeley.xtrace*
- *depends edu.berkeley.chukwa_xtrace.CausalGraph edu.berkeley.xtrace.reporting*

Once we have that, we transform the data into an input format for our visualization tool (JSON), and we end up with an interactive graph, first at higher level of abstractions, then as the user zooms in, the graph transitions into lower levels of abstraction. The format will be similar to a dependency wheel like *figure 2*.