Visualizing Project Evolution Through Abstract Syntax Tree Analysis

Michael D. Feist, Eddie Antonio Santos, Ian Watts, Abram Hindle
Department of Computing Science
University of Alberta
Edmonton, Canada
{mdfeist,easantos,watts1,hindle1}@ualberta.ca

***Abstract*—What is a developer's contribution to a repository? By only counting commits and number of lines changed, existing tools that visualize source code repositories (such as GitHub's graphs) fall short on showing the effective contributions made by each developer. When many commits are viewed as a group, the details are lost. Commit information can be misleading since lines of code give no indication of what was actually being worked on without careful examination of the changed code. Providing a semantic view of this information could provide deeper insights into how software projects evolve since changes to design and features are not clearly visible from line changes alone. We present TypeV: a method for visualizing Java source code repositories. Instead of counting line changes in a commit we extract detailed type information over time by using the differences between abstract syntax trees (ASTs). We are then able to track the additions and deletions of declarations and invocations for each type. Furthermore, we can track each author's type usage over time. Using TypeV, we examine specific cases in well-known repositories where our tool reveals interesting and useful information. We then compare type coverage information from the AST compared to file coverage to determine if unique information is provided by type information.**

I. Introduction

Visualizations can assist our understanding of a software repository. Popular source code repository hosting websites such as GitHub [1] and BitBucket [2] often provide a way to visualize project evolution. These visualizations often show changes to lines of code, as line changes are what the underlying version control system tracks. The advantage of visualizing line changes is that it works for any text-based file. There is a major disadvantage, however: by only visualizing changes to lines of code, GitHub (Figure 1) and similar visualizations lose the *meaning* behind those changes. Although a programmer may type individual characters into their text editor, the lines of code represent something much deeper. A Java programmer, for instance, writes lines of code with the *intent* of creating packages, defining classes, declaring fields, and implementing methods. An author commits lines of code to implement features, write tests, fix bugs, and refactor code bases. Lines of code merely tell us that changes were made; not the *reason* for the changes.

To address the lack of semantically-aware visualizations we created **TypeV**. Instead of visualization changes to lines of code, we visualize changes to type usages and invocations. By using types instead of lines of code, our visualization can better answer the high-level questions programmers often ask [3]. TypeV is composed of two parts: a data extraction application that performs commit-by-commit AST differencing to extract type changes; and an interactive web visualization that allows for the exploration of a project's evolution. Using TypeV, we seek to answer the following research questions:

RQ1: How can we extract type and method usage from repositories over time?

RQ2: How does type coverage compare to file coverage?

RQ3: What extra insight can we gain from type-based visualization?

II. Related Work

We examined research which analyses software repositories for information on language and feature usage, tools for visualizing project evolution, collaboration in software repositories and other tools for understanding repositories.

A. Analysis of Languages and Features

Analyzing source code is a popular research topic and Java is often chosen as the subject of study due to its popularity and abundance of tools. Grechanik *et al.* [4] examined the structure of Java programs mined from 2080 programs. This paper examined the breakdown of syntactic structures in open source repositories, however it does not consider per author statistics. Parnin *et al.* [5] mined repositories to see how Java generics have been used in open source projects and found that generic usages were often introduced by a single developer in a project and were primarily being used for collecting and traversing lists of objects.

Further analysis of the Java language has been done using ASTs. Lämmel *et al.* [6] used ASTs to examine the usage of APIs in Java projects. They found which APIs are popular and if they were used in a framework-like manner. Dyer *et al.* [7] mined AST nodes to study the use of new Java language features over time. The authors found the most popular features and the adoption rate of new features over time. They did not check to see how much a developer used a feature but instead checked to see if they used it at all. In a

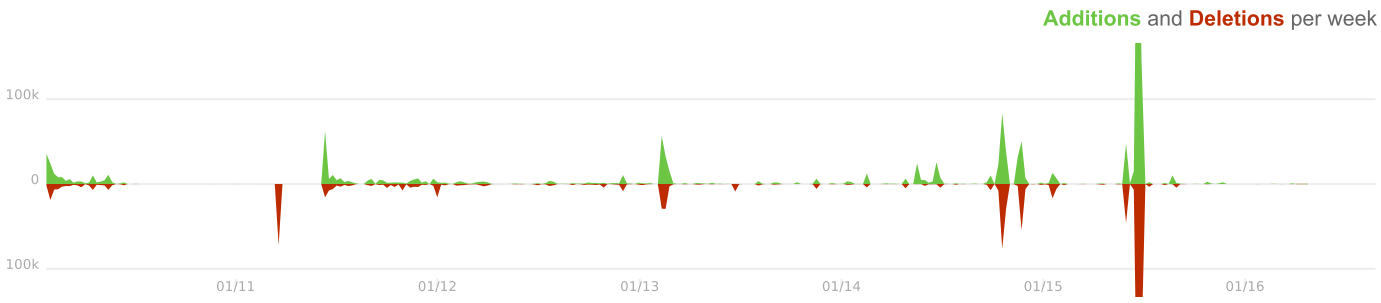

Fig. 1. GitHub's code frequency graph for ANTLR4. Note that the graph was cropped to better show the details. Thus, the changes that occur in mid 2015 are much larger than shown. The additions reach over 400k and the deletions are over 300k.

more general analysis of programming languages, Meyerovich and Rabkin [8] surveyed developers and examined repositories to learn about language adoption and usage. Developers feel that certain language features are more important than others.

The focus of these papers was on how programming languages and features are used by looking at many repositories. Java is a statically-typed language—its types are checked at compile time—yet there is no focus specifically on how types are used throughout a project. Type data may provide additional information on how developers use languages but this has yet to be explored.

B. Visualizing Project Evolution

Several visualization tools have been created to understand how a software system changes over time. Wu *et al.* [9] explored punctuated changes of software repositories with a tool that generates an Evolution Spectrograph. In this visualization, each file has a row in the chart and each cell represents a section of time where an incoming or outgoing dependency can be changed. The colour is determined by the frequency of change. This work was continued to show file directories and display the cardinality of a developer's commits in subsequent paper by Wu *et al.* [10]. The cardinality of a commit is the number of files or subsystems it affects in a system, indicating the scope of the change. Other tools also look at changes in code dependencies [11] or files [12], [13], [14] over time. Alcocer *et al.* [15] created a Performance Evolution Blueprint, which displays the changes in performance over time as a project evolves. Ogawa and Ma created `code_swarm` [16], a generative art visualization which presents the progressive evolution of file changes made by authors in a source control repository over time. Gource [17] is a similar visualization, showing an animated force-directed tree of the file hierarchy as authors make changes to files in the source code repository over time. Ens *et al.* created ChronoTwigger to visualize the co-evolution of source and test files [18]. This allowed testing practices to be more closely examined. Anslow *et al.* [19] created SourceVis, a suite of software visualizations including visualizations for semantically-aware project evolution. SourceVis displays the evolution of Java source code metrics such as number of packages, classes, methods, and fields.

Tools also exist which track changes over time in a more complex manner. Kim *et al.* [20] built a tool called Logical Structural Diff (LSdiff) which examines code changes and produces logical rules to represent them. Changes which do not match the rules are then pointed out to the developer. Reiss and Tarvo created an IDE which allows users to see the history of files and changes to lines of code from each author as they are editing a project [21]. Van Hees and Hage use a voronoi treemap to visualize changes in class and method structure [22]. Servant and Jones created CHRONOS, a tool which creates a timeline of changes to a selection of code as well as the ability to query those changes [23].

C. Project Collaboration

There is also research on how developers use software repositories to collaborate. Wagstrom *et al.* [24] explored the roles that developers take in networked, social development environments like GitHub. These roles were defined by their level of contribution as well as the types of issues they dealt with. They found that developers will take on multiple roles in a project and will sometimes fulfill the same role across different projects. Lee *et al.* [25] developed a tool for visualizing the branching structure of Git repositories. This was used to analyze concurrent workflows in popular open source repositories. Elsen also created a visualization for branches in Git, with the ability to expand branches to view the directory structure of a project at that point [26]. Shrestha *et al.* made a visualization for the geographic location of a commit [27] and Minelli *et al.* [28] looked at the real time changes developers made to a repository.

D. Non-academic works

Some visualization tools are made for managing or making decisions about software rather than studying the changes taking place in an academic context. EasyBi [29] is a service that provides additional visualizations for Git repositories for the purpose of business decisions. EasyBi works by taking the `git log` file and generating charts and graphs from it. It tries to solve the issue of summarizing many commits and has greater controls for granularity and authors but still operates on commits and line differences. The Gitinspector project [30] is an open source tool for the statistical analysis of Git repositories. It provides an advanced timeline view of changes per author and cumulative statistics but again relies

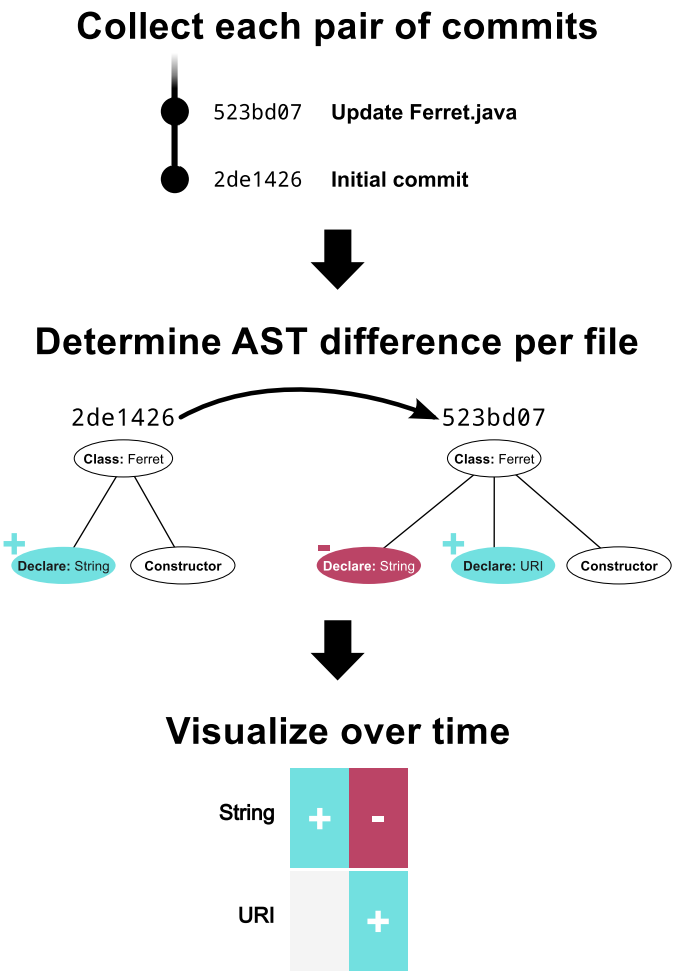

Fig. 2. Data extraction pipeline for TypeV. First, we list the commits in branch chronological order. Then we calculate the AST difference between each changed file across each pair of consecutive commits. Finally, we may visualize the information in a variety of ways.

upon the line differences to summarize the work of each author.

Developers using the massively popular GitHub website for hosting their Git repositories instantly have access to basic visualizations, known as GitHub Graphs (Figure 1) [1]. GitHub Graphs tracks contributions based only on the number of commits and number of lines changed. This has the advantage of not relying on the code which was committed. It does not matter what language is used or if errors exist in the commit—GitHub will simply output the number of line additions and deletions. The issue with this type of analysis is that it does not give a clear idea of what is being changed, only that changes have occurred. This data may be lacking, or even misleading which contributes to the need for better tools.

Of the services we have surveyed, none of the tools that visualize project evolution address semantic history extraction using granular type analysis per author via commit-by-commit abstract syntax tree differencing.

III. Data Extraction

RQ1: How can we extract type and method usage from repositories over time?

Given a Git repository, we want to extract per-author use of types over time. We accomplish this by computing the differences between abstract syntax trees of every consecutive pair of commits (Figure 2). This extracted data can then be used in a variety of visualizations as discussed in Section IV. The reason for using ASTs is that they capture the semantics of the code far better than simple line counts.

An *abstract syntax tree* (AST) is a data structure derived from source code that breaks its syntactic constructs into a tree. Each node in the tree represents a syntactically valid chunk of code. This captures the structure of the code in an abstract and tractable manner. When a programmer makes a non-trivial change to a source code file, that difference will be reflected in the AST. Non-trivial changes are those that affect how a compiler will interpret the source code, and may have effects on the executable code. Trivial changes are those that have no effect on how a compiler will interpret the source text, such as changing insignificant whitespace or comments in source code. Thus, ASTs of consecutive revisions in a code repository can be compared to see what *structures* a programmer is modifying.

The Java ASTs in our analysis were generated using Spoon [31]. Spoon is a tool for transforming and analyzing Java source code. It breaks code up into a meta-model consisting of three parts: structural elements, code elements and references to program elements. According to Spoon's authors: "The structural part contains the declarations of the program elements, such as interface, class, variable, method, annotation, and enum declarations. The code part contains the executable Java code, such as the one found in method bodies. The reference part models the references to program elements (for instance a reference to a type)" [31]. The Spoon model is convenient because it provides the infrastructure for performing AST differencing (described later) while retaining code elements for analysis afterwards. Spoon also preserves all type and package (library) information, information vital to our semantic analysis.

To determine what has changed between two subsequent commits, we computed their *AST difference*. Given two ASTs, an AST difference (or simply as *AST diff*) states which nodes must be added and which nodes must be removed to transform one tree to the next (step two in Figure 2). This allows us to precisely track changes between two consecutive revisions of the same file. For AST diffing, we used GumTree [32], an algorithm made to compute the difference between two ASTs generated by Spoon. For new files and deleted files, it is unnecessary to calculate an AST difference; we simply run Spoon on the files and treated everything as an addition or deletion respectively.

Once we calculate the AST diffs, we were able to traverse through the trees to count the number of added or deleted *declarations* and *invocations* for a given type.

declaration: Stating the use of a field or variable of a given type. For example declaration of a `String` type variable might look as follows:

```
String myString;
```

invocation: A call site for a method that belongs to a type. For example an invocation of the trim method of a String type might look as follows:

```
myString.trim();
```

Each change of declaration or invocation is detected and tallied per file per commit. These changes are emitted in an ad hoc file format. For example, the following is the output of a commit that changed two declaration of `String` variables to `URI` variables:

```
#DECLARE | INSERT | java.net.URI | 2
#DECLARE | DELETE | java.lang.String | 2
```

This states that two declarations of type `java.net.URI` were added, and two declarations of type `java.lang.String` were deleted. Note that TypeV is able to find the fully qualified type names. Thus, the source text may not literally have the characters `java.net.URI`, but Spoon was able to infer from the import statements in the Java source text that references to "URI" resolve to the fully-qualified type name `java.net.URI`.

In order to gather all the changes for a given repository over time, we generated ASTs for each commit and compared the difference. We did this by running `git log` over the Git repository and listed all commit IDs in chronological order (Figure 3). For each commit, we collected the following information, as can be seen in Figure 4: the author's name, date of commit, commit ID (SHA), commit message, files the commit edited, and files in the repository at the time of the commit. For all the edited Java files we also compared the file at the time of the commit with its parent commit using AST differencing as described above. We achieved this by invoking `git difftool` and specifying our AST differencing tool. One case where `git difftool` fails is when the commit has no parent, such as the initial commit. We were able to handle this by calling `git show` and saving the output to a temporary file to be used with the AST differencing tool. When calling `git show` instead of `git difftool` we treat everything as an addition, as in commit A in Figure 3. Since we compare each commit to its parent as reported by Git, we are able to account for any branching that happened outside the primary branch of the repository (usually named `master`). If a commit has more than one parent, like commit D in Figure 3, it indicates a merge commit in Git.

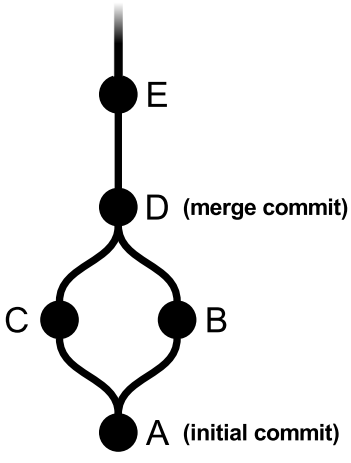

Fig. 3. Commits increase in time from bottom up. Therefore, ASTs from B and C are diff'd against ASTs in A. ASTs from E are diff'd against D. Since D is a merge commit, its changes are ignored as they are recorded in C and B.

We deliberately chose to ignore merge commits. A merge commit has two cases to consider: The first case is when the parents of the commit can be automatically merged. In this case all the relevant changes are already recorded in the parent of the commit so we can safely ignore the merge commit [33]. The second case occurs when the parents cannot be merged automatically: this is a *merge conflict* which requires manual resolution on the part of a developer. In this case, the developer will try to resolve the issue by modifying code. If we decide to include the merge commit we run into several edge case issues such as how to compare the merge commit with its multiple parent commits and how to account for the possible modifications made to resolve the merge conflict. We could consider comparing the merge commit with only one parent commit at a time; however, we will end up counting all the changes already recorded in the other parent branch. Additionally, all changes made during the resolution of a conflict will be attributed to the author making the merge. This may exclude the author that actually wrote the code in the first place. The other option is to simply ignore the merge commit such that we do not count the changes done during a merge commit. We made the assumptions that merge conflicts occur infrequently and when they do occur, the changes made to resolve a merge conflict are usually minor. Therefore, we chose to ignore merge commits.

The data collected from running the AST differencing component over the commits is saved to a file to be visualized later; this data is in a general format, and is not bound to any particular visualization. Figure 4 summarizes the hierarchy of the data extracted by our tool.

The major downside of AST diffs over a method like line count is that ASTs are more expensive to compute. Our experiments were run on an Intel i7 with 16GB of RAM and an SSD. Calculating the AST diffs varied from an hour or two for smaller projects to a couple of days for very large projects. For example ANTLR4 took about 8 hours while Apache BookKeeper only took a couple of hours. The good news for ongoing projects is that AST diffs only need to be computed for new commits; TypeV can resume calculating diffs and augment statistics at any time.

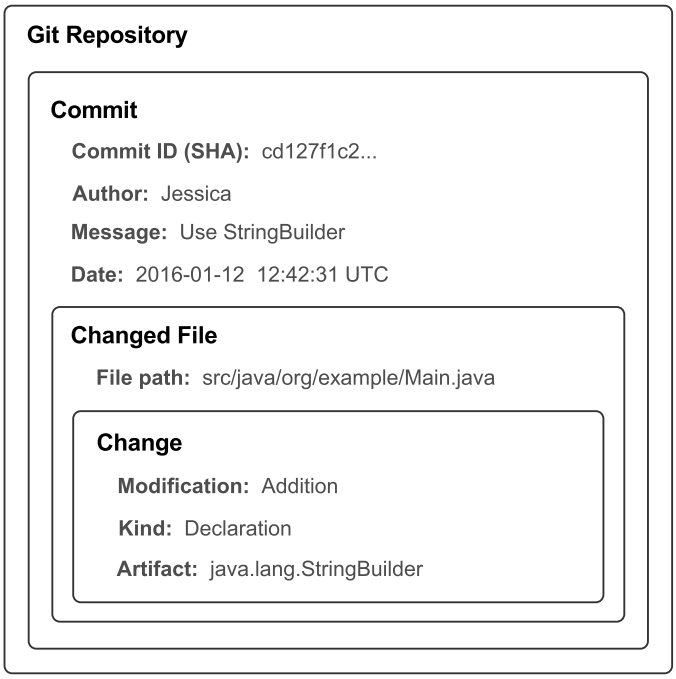

Fig. 4. Hierarchy of data extracted by TypeV. Each repository may have zero or more commits; each commit may have zero or more changed files; each changed file may have zero or more changes, as computed by our AST differencing tool.

IV. Visualization

Although rich in detail, the AST diffs are difficult to analyze if displayed in their raw form; hence, we created an HTML5 web visualization application which is intended for developers, project managers, and researchers to use (Figure 5). The source code for TypeV—both the AST differencing component and the web application—is available online on GitHub.[1]

A. Timeline View

For our main demonstration of semantically-aware project evolution, we chose an interactive timeline view that displays the most popular types over time. The timeline displays several rows of stacked bar graphs which show changes to an individual type or method over a selectable period of time in a project (Figure 5). The light blue and dark red portions account for the number of additions and deletions respectively, in a similar fashion to GitHub's code frequency graph (Figure 1). The height of each bar shows the relative number of changes to a type. The width of each bar is fixed, and represents a time period for binning commits that can be configured to an hour, day, week, or month's worth of commits per each bar. A bar that completely fills its row vertically is the period with the largest number of changes to that type in the visible date range. The height of the other bars in the row are in proportion to the maximum number of changes. The bar height is calculated by multiplying the maximum bar height by the number of changes during that time period divided by the maximum number of changes attested for this type.

One can interact with the overview by hovering and clicking on the bars. When a user mouses over a specific bar in the graph they are given summary information for the type and time period at that location. The information includes the fully-qualified name of the type, the number of commits, the number of authors, the number of additions, the number of deletions, and the start and end date of that time period. If the user clicks on the bar, they are given more detailed statistics which list the authors who made the commits and each commit message.

To allow for a more detailed analysis of a specific component of the repository we added filtering tools. Users can filter by type name; view changes that fall within a specific start and end date; and select whether to see only declarations of types, types used in declarations and invocations, or only invocations.

One issue with analyzing source code repositories is that authors can commit from a variety of different computers which can have different emails and usernames attached to their Git configuration. This may result in one author appearing as several different authors in the `git log`. There is no perfect way to deal with this problem; however, we added an interface for managing authors (Figure 6). A user can manually specify an author's various aliases using a simple dropdown list. All statistics attributed to an author's various aliases are then remapped as belonging to one canonical author as specified by the user. The author managing view also provides a facility for selecting and deselecting authors shown in the timeline. Thus, visualization users can focus on the changes made by a particular author or a group of authors, rather than having to view the changes made by all authors.

B. Design Decisions

The purpose of displaying stacked bar charts is that they are a familiar visualization. They can be viewed as an enhanced version of GitHub's code frequency graph [1], with semantic data. We chose to create a similar visualization to GitHub to work with a layout people may already be familiar with and to evaluate our visualization tools' density of data.

The stacked bar graphs succinctly displays the relative amounts of additions and deletions per each time period. This creates a dense view of the project's evolution. We chose to make the height of the bars relative to only the type itself to make it easy for a user to scan a single type and see when it was modified. This makes tasks such as finding a refactoring (Section VI-A) very clear to see simply by browsing the line. We stacked the additions on top of the deletions such that users can immediately compare the relative frequency of changes of a single type throughout the chosen time span.

Our choice of colours was motivated to be high contrast and culturally indicative of their meaning. The particular colours chosen are high contrast, and are distinct to those experiencing protanomaly or deuteranomaly—red-green colour blindness. The red colour suggests a negative valence [34], hence sug-

[1] https://github.com/mdfeist/TypeV

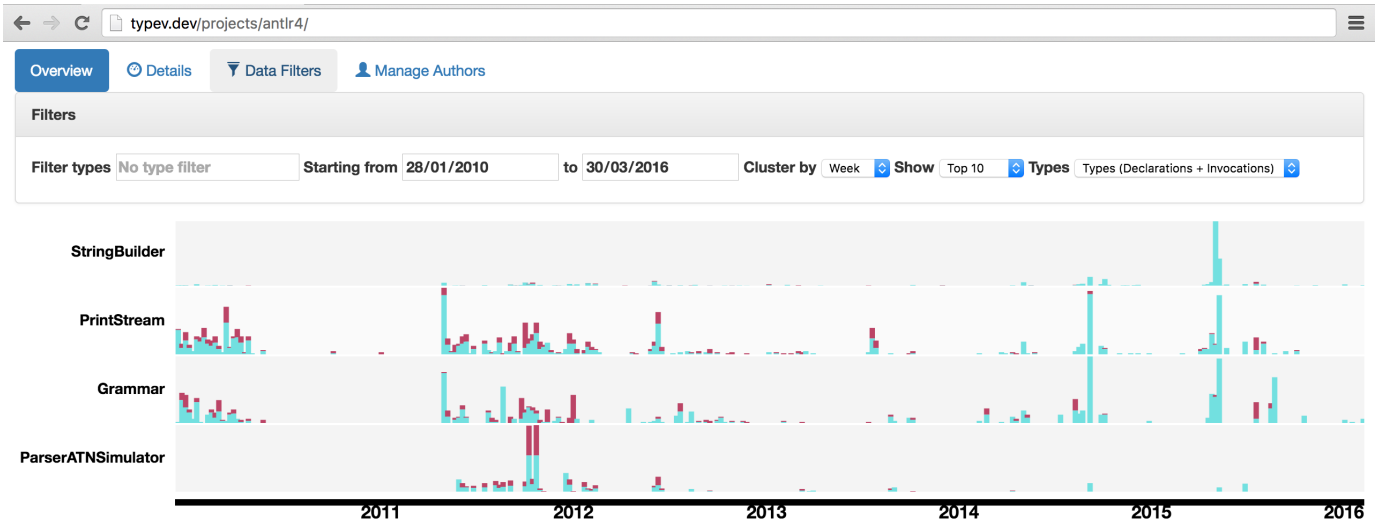

Fig. 5. Screenshot showing the main view for TypeV. The data being displayed is from the ANTLR4 repository. Light blue bars indicate **additions**; dark red bars indicate **deletions**.

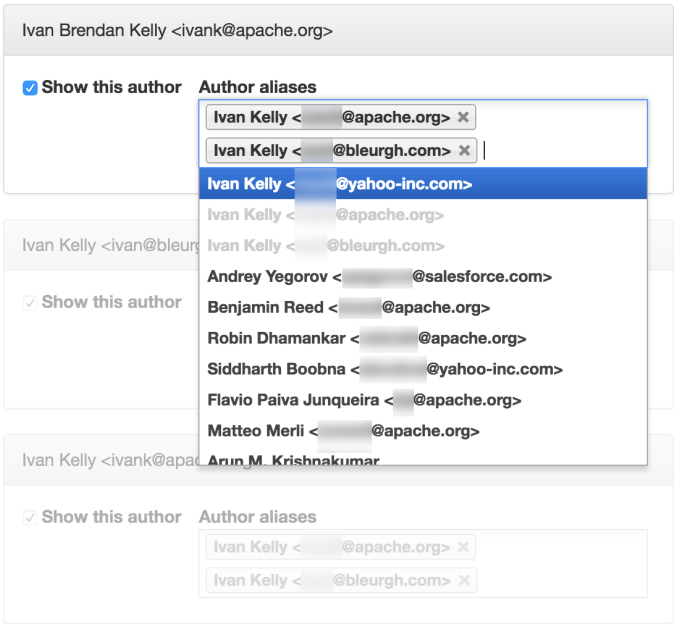

Fig. 6. Screenshot showing the author management view for Apache Bookkeeper. Users can use a drop-down list to select the aliases for an author, and select whether to show or hide the author in the timeline view. Authors in the drop-down list are sorted in decreasing order of Jaccard similarity.

gesting deletions. The blue serves as a contrast to the red, indicating the opposite action (in this case, additions).

The rows are sorted so that the graph for the most used types or methods appear the top and are in descending order of use such that the most frequently used type is the first visible. The height allows users to see when certain types are being edited and ensures that highly used types do not overpower less used types making their changes less significant.

By adding interaction, the user is able to explore the timeline to find the specific information that they are interested in and drill-down at their leisure. Although our tool extracts a great deal of information, users can gradually reveal more information as they need it, rather than be bombarded with all of the information at once.

TABLE I
SUMMARY OF PROJECTS USED IN THE EVALUATION.

Project	Commits	Authors	Files	Types	Changes
ANTLR4	3930	19	1792	2635	58425
BookKeeper	660	10	758	1872	31817
Curator	1524	35	1211	1269	27957
Tika	860	10	604	757	20764

Totals for the four projects we analyzed. "Changes" refers to the total number of invocation or declaration changes observed from the AST diffs.

V. EVALUATION

TABLE II
COMPARISON OF TYPE COVERAGE AND FILE COVERAGE IN PROJECTS

Project	Wilcoxon p-value	Pearson (linear) Correlation	Spearman (rank) Correlation
ANTLR4	$p \ll 0.01$	0.943	0.907
BookKeeper	0.313	0.984	0.972
Curator	0.007	0.975	0.980
Tika	0.653	0.955	0.998

RQ2: Is it worth the effort of computing commit-by-commit AST differences if we can simply count changes to files? We will answer this question by comparing type coverage and file coverage. We define *type coverage* as the number of types that have been touched out of the total number of types during either a certain time period or the entire project. *File coverage* is the same measurement, but using the number of source files touched instead. File coverage can be trivially determined

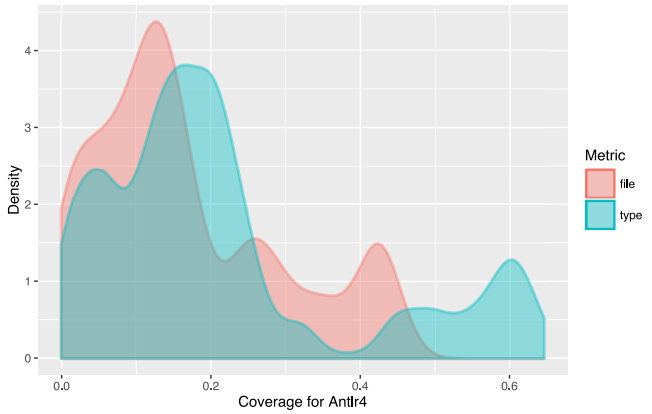

Fig. 7. R density plot of type and file coverage for commits. The graph shows what percentage of type and file coverage the majority of commits have. One can see that the majority of commits have a type and file coverage around 0% to 20%. We can also see a difference between these measurements on the higher end of the graph. File coverage seems to have a group of commits that had a file coverage between 20% and 50% whereas type coverage have a similar looking grouping but between 40% and 60%.

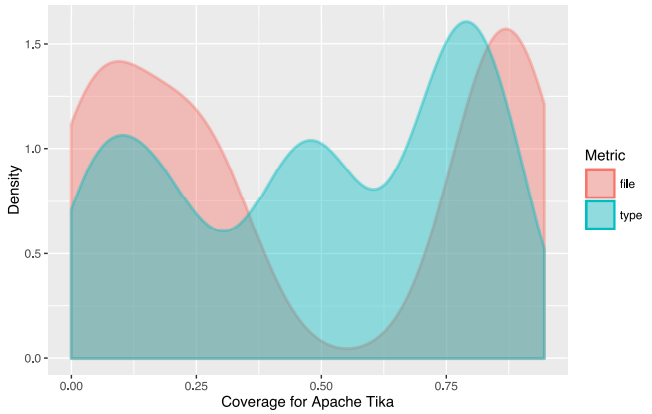

Fig. 8. R density plot of type and file coverage for commits. Here we can see that commits did not have much file coverage between 25% and 75%. On the other hand there were a lot of commits that have type coverage around 50%.

from an author's commit history. Type coverage, however, is uniquely provided by AST differencing, as we have described.

Java's syntax encourages the creation of classes, thus we expect substantial Java projects to define many domain-specific types. Additionally, Java places the constraint that each source file must define one class. When writing or contributing to a project, the author's choice of which part of the program to edit will in turn affect what types they end up working with — that is, it will affect the author's type coverage. Similarly, the set of files that an author works with affects that author's file coverage.

To determine if type coverage yields different information than file coverage, we analyzed four open source Java projects: ANTLR4, Apache BookKeeper, Apache Curator and Apache Tika (Table I). We chose ANTLR4 because it is a large Java project with a rich history of commits. ANTLR4 is also a member of the Qualitas Corpus [35], a curated collection of Java systems curated for empirical software engineering research. We then collected three smaller projects from the Apache foundation, as they host a large number of well-curated open source Java projects. These particular Apache projects were chosen for their clear development practices and easily available documentation. We counted the file coverage and type coverage on a commit-by-commit basis for each author. That is, each datum represents a commit with three dimensions: file coverage, type coverage, and its commit date. Data is sorted in ascending order of commit date. Table II lists a summary of the statistics measured.

We displayed the density distributions for two different projects: ANTLR4 (Figure 7) and Apache Tika (Figure 8). These plots were produced using R and show the distribution of the amount of commits that have X% of type and file coverage. The red plot represents file coverage and the blue plot represents type coverage. The x-axis shows the percent of the coverage and the y-axis shows the number of commits which have the similar percent of coverage. For example, Figure 8 shows that there are many commits with 50% type coverage and relatively few commits with 50% file coverage. Also notice how in Figure 7 the type coverage is pushed to the right, meaning commits usually touch a higher percentage of types than files. So if we only looked at files touched we might be missing important changes in the project. Figures 7 and 8 show that even though the type and file coverage plots may have a similar shape, the commits are showing different levels of information.

In general, type coverage and file coverage are strongly correlated, both using Pearson (linear) correlation, and Spearman (rank) correlation. As type coverage and file coverage are both cumulative measures, positive correlation is expected. However, even when the distributions are significantly different (as with ANTLR and Curator), the correlation is still strongly positive, but not 100%. This means that when the coverage for types and files of a commit increase, the relationship is not necessarily one-to-one. This is remarkable in a language like Java, as classes and files are generally in a one-to-one relationship.

We conclude from these findings that type coverage yields related, yet different information than file coverage alone.

VI. Illustrative Case Studies

RQ3: What extra insight can we gain from type-based visualization? We present case studies demonstrating the insight AST differencing can provide.

A. ANTLR4

We chose ANTLR4 because it is a substantial Java project with a commit history spanning several years. We can also see distinct changes throughout its history.

For example, in December of 2011, in Figure 9 we can see that the `ParserATNSimulator` class has a large amount of additions and deletions in roughly equal proportion. This suggests that `ParserATNSimulator` is not being removed

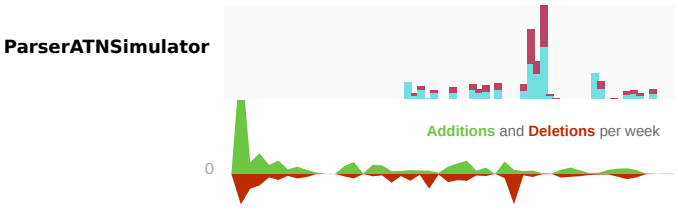

Fig. 9. Screenshots of TypeV (above) and GitHub's code frequency graph based on line difference (below). TypeV is showing changes to `ParserATNSimulator` in ANTLR4 while GitHub's graph is showing all line changes to ANTLR4. The date ranges from approximately April 2011 to April 2012. Note that both the TypeV and GitHub graphs are approximately lined up along the x-axis (time) and that we can see the loss of detail in the GitHub graphs.

from the project but instead there is some refactoring being done. If we take a closer look at the commit messages we can see that they mention refactoring and reorganization of code. Zooming in on that date, shows that during that month they were finishing work on a new version of `ParserATNSimulator`.

If we tried to do the same analysis with the GitHub tools it would be significantly more difficult to reach this conclusion (Figure 9). First, if we only look at line changes this change to `ParserATNSimulator` is overwhelmed by other changes and hardly shows up on the GitHub tools. If we did notice an increase in activity we would not know what types were being worked on. Finally, we could not quickly look at the commit messages for that specific time and type if we used the GitHub tools.

Using TypeV we can see that at certain times there were major changes across all types. This shows us when major changes to the code were being done versus code maintenance.

B. Apache BookKeeper

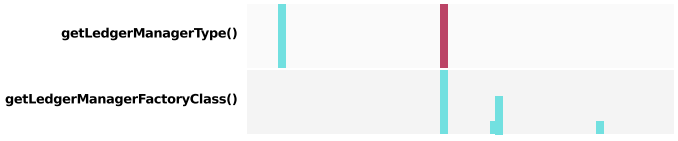

Fig. 10. Shows deprecated invocation `getLedgerManagerType()` and replacement invocation `getLedgerManagerFactoryClass()` for Apache Bookkeeper.

The purpose of *deprecating* software features is to give developers time to change projects to follow a new or better standard without breaking existing code. When a piece of code becomes deprecated, continuing its use may lead to design flaws or even security vulnerabilities. For example, in Apache Nifi, an earlier version of the software used a compromised key-derivation function; hence, this class was deprecated, but left in for compatibility with older versions of the software [36]. Viewing changes in terms of types can show when usage of deprecated classes are added or removed from a project as well as how many remain over time. Given the time at which a feature was deprecated, seeing the progress of replacement has some important implications. In the case of a security flaw, that time could indicate times at which systems or its users were vulnerable. If a feature is marked for removal, a third party using the deprecated feature has a limited time to make the appropriate changes. Removing all instances of deprecated features could also mean a project can be upgraded to use new tools or features that were not possible when the deprecated features were present.

We searched for Apache foundation projects that featured an instance of deprecation and found Apache BookKeeper. As an Apache Foundation project it is a well-curated example of an open source software project.

Apache Bookkeeper deprecated the method AbstractConfiguration.getLedgerManagerType() in release 4.2.0[2] and replaced it with AbstractConfiguration.getLedgerManagerFactoryClass(). If we look at getLedgerManagerType() we can see at the time of the start of the deprecation all instances were removed and getLedgerManagerFactoryClass() was added (Figure 10). After the deprecation, we see no more changes to getLedgerManagerType() and only some additions of getLedgerManagerFactoryClass().

VII. Discussion

Having tools for analyzing repositories based on type usage and method invocation benefits many parts of the software development process. Software managers require a high level understanding of the state of a repository and the changes being made which affect the functionality of the software. Tracking developer contributions in a less misleading way is also important for management since important contributions may be small and focused, but nevertheless, significant. Line differences only provide meaningful information when they are analyzed by someone with familiarity with the source code being changed. Providing tools which bring more meaning than line differences would allow people less familiar with the code to understand changes being made. When large changes are made that cover many aspects of a project, it is more useful to see how the change affects the software rather than noting that several thousands of lines have changed. Seeing how a project has changed over time is also vital for management since evaluating the state of completion of software can be very difficult.

The timeline visualization presented in this paper (Section IV) is intended to be a practical tool that enables users to investigate the evolution of a project. Our timeline view can be seen as an evolution of simple visualizations such as GitHub's code frequency [1]. GitHub's graph is static, however it allows one to see an entire project's history at a glance. TypeV improves this by enabling one to see a particular *type's* history at a glance. Given the greater density of information available, we made our timeline interactive such that a user is able to drill-down and explore the data at their own pace. Using interaction, TypeV not only shows what types have changed in a project, but makes it possible to see *why* a type

[2]https://bookkeeper.apache.org/docs/r4.2.0/apidocs/deprecated-list.html

has changed. We believe conventionality is a strong point in a developer's goal to make interesting discoveries in a project's evolution, but this has to be confirmed in user studies.

TypeV can also be useful for bug triaging. Bug triaging for large open source projects is a very time consuming task [37], with hundreds of bugs being reported daily. Each bug must be processed and assigned to a developer that has the experience required to fix the bug. In order to be effective, the triager has to have knowledge of what each developer has worked on as well as the parts of the code that may be affected by the bugs. With TypeV, the triager is able to directly see what types a developer has worked with over time. This is important information for triaging since the type causing the bug, if known, will have a list of associated developers who have recently used that type and can be assigned the bug. The commit messages for the affected types are also visible, providing greater context about how the type has been used. The assigned developer can also use type information to locate other pieces of code affected by a bug and know which developers have been using the affected types.

A. Threats to Validity

Internal threats to validity include our treatment of merge conflicts. As mentioned in section III, we ignore all merges, so if there is a merge conflict, it is possible that a developer made non-trivial changes. Changes made during merge conflict resolution are missed by TypeV. We have made the assumptions that if this case happens, the changes are minor.

Some actions in Git present a problem gathering the statistics. Authors can pull other projects into the repository and these additions will count towards the author's changes. These will show up as massive changes by a single author with many types that they did not actually create. Changes of this size are visible in the graph and can be found by inspecting the commit messages.

In addition, the author statistics given in Table I was calculated after manual author merging. We are not personally familiar with the project, let alone the authors collaborating on the project. Therefore, our efforts to manually merge authors may have been erroneous.

An external threat to validity is the limited number of projects used in the validation. We picked curated projects that had a long development cycle and looked into some case studies. However, because of the limited number of projects we observed, our conclusions may not hold for the majority of software projects.

B. Future work

A user study is needed to test the usefulness of TypeV. Some questions that need to be answered are: Can users new to a project easily find when major changes took place and, more importantly, what was being changed? Given a specific time period, can users find what was being worked on? The study should test TypeV against a line-based visualization such as GitHub's graphs [1] and a file-based visualization such as `code_swarm` [16].

The methodology used in TypeV is only well-defined for statically-typed languages such as Haskell or Java; it is not well defined for dynamically-typed languages such as Erlang or JavaScript. However, type annotation systems such as TypEr for Erlang [38] or statically-typed language dialects such as TypeScript for JavaScript may prove to be sufficient to enable type analysis as we have demonstrated for Java.

A weakness of the timeline view described in this paper is that details about an author's activities are buried. As future work, the data extracted by TypeV can be used to create a visualization similar to Gource [17] or `code_swarm` [16]. These visualizations animate authors moving back-and-forth over files they edit; we proposed to animate authors' activities over *types*. The focal point would be a particular type. The type itself may be represented abstractly as a shape or its UML representation that grows and shrinks as its usage increases and wanes over time.

VIII. Conclusion

In this paper, we introduced TypeV, a code repository analysis and visualization tool. Using commit-by-commit abstract syntax tree differencing enables insightful visualizations that are not captured by tools that use commit counts, line counts, or file changes alone. In this paper, we addressed the following research questions:

RQ1: How can we extract type and method usage from repositories over time?

> Using abstract syntax trees, we can determine the types affected by each commit in a repository. Type information is not directly available through regular line diffs recorded by Git.

RQ2: How does type coverage compare to file coverage?

> Type coverage and file coverage are found to be highly correlated but do not always have the same distribution or a one-to-one ratio as might be expected in Java.

RQ3: Does type information give any interesting or useful insight into the project?

> The evolution of a project and developer activity can be seen through the changes to types in a project. Significant events can be observed with greater understanding by seeing the changes to types in addition to existing commit information.

Although calculating ASTs can be expensive, it is not computationally infeasible, and we benefit from semantically-aware code evolution visualizations that can be used to gain better insight into our software.

REFERENCES

[1] GitHub Inc. About Repository Graphs - User Documentation. [Online]. Available: https://help.github.com/articles/about-repository-graphs/

[2] Atlassian Pty Ltd. Awesome Graphs for Bitbucket: visualized statistics for Git and Mercurial repositories. [Online]. Available: https://blog.bitbucket.org/2015/08/05/awesome-graphs-for-bitbucket-visualized-statistics-for-git-and-mercurial-repositories/

[3] J. Sillito, G. C. Murphy, and K. De Volder, "Questions Programmers Ask During Software Evolution Tasks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. ACM, 2006, pp. 23–34. [Online]. Available: http://doi.acm.org/10.1145/1181775.1181779

[4] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, "An empirical investigation into a large-scale java open source code repository," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 11:1–11:10. [Online]. Available: http://doi.acm.org/10.1145/1852786.1852801

[5] C. Parnin, C. Bird, and E. Murphy-Hill, "Java generics adoption: How new features are introduced, championed, or ignored," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 3–12. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985446

[6] R. Lämmel, E. Pek, and J. Starek, "Large-scale, ast-based api-usage analysis of open-source java projects," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. New York, NY, USA: ACM, 2011, pp. 1317–1324. [Online]. Available: http://doi.acm.org/10.1145/1982185.1982471

[7] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of ast nodes to study actual and potential usage of java language features," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 779–790. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568295

[8] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 1–18. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509515

[9] J. Wu, C. W. Spitzer, A. E. Hassan, and R. C. Holt, "Evolution Spectrographs: visualizing punctuated change in software evolution," in *7th International Workshop on Principles of Software Evolution, 2004. Proceedings*, 2004, pp. 57–66.

[10] J. Wu, R. C. Holt, and A. E. Hassan, "Exploring software evolution using spectrographs," in *11th Working Conference on Reverse Engineering, 2004. Proceedings*, 2004, pp. 80–89.

[11] R. G. Kula, C. D. Roover, D. German, T. Ishio, and K. Inoue, "Visualizing the evolution of systems and their library dependencies," in *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, Sept 2014, pp. 127–136.

[12] A. Hanjalić, "Clonevol: Visualizing software evolution with code clones," in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, Sept 2013, pp. 1–4.

[13] M. Burch, T. Munz, F. Beck, and D. Weiskopf, "Visualizing work processes in software engineering with developer rivers," in *Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on*, Sept 2015, pp. 116–124.

[14] L. Voinea and A. Telea, "An Open Framework for CVS Repository Querying, Analysis and Visualization," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. ACM, 2006, pp. 33–39. [Online]. Available: http://doi.acm.org/10.1145/1137983.1137993

[15] J. P. S. Alcocer, A. Bergel, S. Ducasse, and M. Denker, "Performance evolution blueprint: Understanding the impact of software evolution on performance," in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, Sept 2013, pp. 1–9.

[16] M. Ogawa and K.-L. Ma, "Code_Swarm: A Design Study in Organic Software Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1097–1104, 2009. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2009.123

[17] A. H. Caudwell, "Gource: Visualizing Software Version Control History," in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, ser. OOPSLA '10. ACM, 2010, pp. 73–74. [Online]. Available: http://doi.acm.org/10.1145/1869542.1869554

[18] B. Ens, D. Rea, R. Shpaner, H. Hemmati, J. E. Young, and P. Irani, "Chronotwigger: A visual analytics tool for understanding source and test co-evolution," in *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, Sept 2014, pp. 117–126.

[19] C. Anslow, S. Marshall, J. Noble, and R. Biddle, "SourceVis: Collaborative software visualization for co-located environments," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, 2013, pp. 1–10.

[20] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 309–319. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070531

[21] S. P. Reiss and A. Tarvo, "Tool demonstration: The visualizations of code bubbles," in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, Sept 2013, pp. 1–4.

[22] R. van Hees and J. Hage, "Stable voronoi-based visualizations for software quality monitoring," in *Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on*, Sept 2015, pp. 6–15.

[23] F. Servant and J. A. Jones, "Chronos: Visualizing slices of source-code history," in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, Sept 2013, pp. 1–4.

[24] S. Wagstrom, Jergensen, *Roles in a Networked Software Development Ecosystem: A Case Study in GitHub*, 2012. [Online]. Available: http://digitalcommons.unl.edu/csetechreports/149

[25] H. Lee, B. K. Seo, and E. Seo, "A Git Source Repository Analysis Tool Based on a Novel Branch-Oriented Approach," in *2013 International Conference on Information Science and Applications (ICISA)*, 2013, pp. 1–4.

[26] S. Elsen, "Visgi: Visualizing git branches," in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, Sept 2013, pp. 1–4.

[27] A. Shrestha, Y. Zhu, and B. Miller, "Visualizing time and geography of open source software with storygraph," in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, Sept 2013, pp. 1–4.

[28] R. Minelli, A. Mocci, M. Lanza, and L. Baracchi, "Visualizing developer interactions," in *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, Sept 2014, pp. 147–156.

[29] "EasyBi business intelligence," https://eazybi.com/features, 2016, accessed: 2016-04-09.

[30] "Gitinspector," https://github.com/ejwa/gitinspector, 2016, accessed: 2016-04-09.

[31] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, p. na, 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01078532/document

[32] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and Accurate Source Code Differencing," in *ASE 2014*, France, 2014, p. 11 p. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01054552

[33] (2016) Git - git-merge Documentation. Git. [Online]. Available: https://git-scm.com/docs/git-merge/2.8.4#_true_merge

[34] A. C. Moller, A. J. Elliot, and M. A. Maier, "Basic hue-meaning associations." *Emotion*, vol. 9, no. 6, p. 898, 2009. [Online]. Available: http://psycnet.apa.org.login.ezproxy.library.ualberta.ca/journals/emo/9/6/898/

[35] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010, pp. 336–345.

[36] The Apache Software Foundation, "NiFi System Administrator's Guide," https://nifi.apache.org/docs/nifi-docs/html/administration-guide.html#key-derivation-functions, 2016, accessed: 2016-04-10.

[37] A. S. Badashian, A. Hindle, and E. Stroulia, "Crowdsourced bug triaging," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 2015, pp. 506–510. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7332503

[38] T. Lindahl and K. Sagonas, "Typer: A type annotator of erlang code," in *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, ser. ERLANG '05. New York, NY, USA: ACM, 2005, pp. 17–25. [Online]. Available: http://doi.acm.org/10.1145/1088361.1088366