Provenance and Algorithmic Accountability in Software Engineering

Heather S. Packer[1]

University of Southampton, Hampshire, SO17 1BJ hp3@soton.ac.uk

Abstract. The frameworks and tools used for software development can provide insights into software engineering practices and the evolution of code. Moreover, many of these frameworks provide RESTful APIs that enable provenance information to be retrospectively modelled. In this paper, we explore whether information from the software engineering process can be used to support accountable algorithms. Specifically, we investigate the Git version control system which has already been modeled in PROV, and layer it with information from Git repositories pertaining to the software development process such as issue tracking and pull requests.

Keywords: Provenance · Software Engineering · Git.

1 Introduction

In an ideal world, algorithms would use provenance to model their inputs and outputs to support their accountability. However, in the real world, this ideal does not happen because of limited resources and skills. The development process creates artifacts that can be leveraged to model a code's development. Concretely, information from Version Control Systems (VCS) and their workflows can be used to build provenance models retrospectively without having developers requiring specialist knowledge.

Building these models could be relatively cheap for developers using supporting tools, however current research has not investigated whether the insights these models could provide would be useful in the terms of algorithmic accountability. For example, it is possible for these models to describe whether an algorithm has been developed following a particular process and who is connected to the evolution of features in a code base. But it is unclear what role this metadata could play in algorithmic accountability.

In order to investigate this, we explore which information can be used to attribute different activities involved with the evolution of a code base to actors from a VCS and other tools. In particular, we focus on Git, a VCS which has already has a tool, Git2PROV, that generates PROV statements about files in a Git repository. We discuss augmenting this PROV model with information from the GitHub API which exposes Git pull requests and issue tracking, which provides information about which code is changed for a particular purpose. Then, we also discuss augmenting the model with information from other tools with RESTful APIs, such as team tools which can provide details about working practices and Continuous Integration (CI) frameworks which can provide information about testing and it's results.

The paper is organised as follows, Section 2 briefly introduces related work. In Section 3 we discuss Git and Git Workflows, followed by Section 4 which describes the information we can enhance Git2PROV models. Then in Section 5, we discuss what how our proposed PROV model can support algorithmic accountability. Lastly, we conclude in Section 6.

2 Related Work

This paper aims to increase the accountability of web-hosted version control systems. There has been some work which looks at the relationship between collaboration and transparency [4], however, despite there widespread use, there is a lack of focus on making this types of systems support algorithmic accountability.

To situate our work, we look at the tool Git2Prov [1] which converts metadata stored in Git version control system into the PROV format [3]. We investigate how this model can be extended with inbuilt and third-party tools for Git web-based hosted services, such as GitHub and BitBucket.

3 Git and Software Engineering

Git manages low-level version control transactions. Git repositories are commonly managed using web-based services which host projects. These web-based repositories, such as GitHub and Bitbucket, provide extra details including deployment strategies and track issues. The evolution of coding projects can vary greatly due to a team's working practices and the range of features Git offers to manage changes, and there is no standardised process on how to interact with Git. A Git Workflow describes the working practices for a project and defines how the flow of changes will be applied. A workflow may include how developers use:

1. **Pull requests** to encapsulate a set of changes to be made to a code base. Web-based repositories provide a user-friendly interface for discussing proposed changes before integrating them into the official project.
2. **Branches** to encapsulate the development of a change or feature.
3. **Merging Protocol** for determining how developers merge their changes, including which branch to merge to and any pre-merge checks that need to be satisfied such as code-reviews.
4. **Testing Protocol** for determining at which stages and which tests should be executed.

This metadata can be used to extend the information model used by Git2PROV, to describe higher level transactions made during the software development process.

4 Inbuilt and Third Party Tools

Web-based hosting services for version control of source code provide inbuilt and third-party tools to support the software development process. These tools broadly fit into the following categories: 1) Code quality and review tools; 2) Continuous Integration; 3) Monitoring tools; and 4) Team tools.

4.1 Code Quality and Review Tools

Code Quality provides tools to support code reviews by teams. They use automated tools, often including static analysis to identify potential issues in the code. In terms of accountability, code quality tools can provide details about potential issues and how they are handled by a team, whether its identification is followed by an issue being added to the issue tracker, or whether there is a flourish of development on a particular feature, or whether there is no action taken. Logging the type of potential issue and the actions that follow it, would allow the practices in different projects and teams to be compared. This would enable some judgment to be made on the quality of the code.

4.2 Continuous Integration

Continuous integration (CI) tools automatically build and test your code as it is pushed it to a repository, preventing bugs from being deployed to production. CI tools are not appropriate for every software project, for example, those that are small or contain untestable code. However for larger projects, CI has benefits such as catching integration issues early, enforcing testing discipline and decrease code review times. The value of CI depends on the coverage, the types, and the quality of tests. In terms of accountability CI tools can provide information about commits, the log providing details about the tests performed for each commit and details about the build.

4.3 Monitoring Tools

Monitoring tools provide real-time metrics and summary reports about the impact of code changes, performance, errors, and analytics. For accountability, these reports can be used to provide more evidence of specific details about the software used. Incorporating these reports into a PROV model, enables the comparison between different versions of the software and could be related to issues in the issue tracker.

4.4 Team Tools

Team tools enable management of teams for a project, and their division into sub-teams to develop features for a project. It enables the developers to manage permissions, and assign particular teams to a project. In terms of accountability, these tools can provide a detailed list of the people on a team and their administrative rights. Logging how teams evolve using PROV alongside information about pull requests and issues can show how it's members are used during software development. The experience of a team member can be inferred from the number of features they have developed, how many times they have reported issues, or how many issues they have fixed. Teams might use different strategies for addressing issues. For example, teams dealing with low-level issues might contain less experienced individuals and choose to assign its experienced developers to high-level issues. Understanding what role and how experienced a team member is could speak to the importance and complexity of features being developed.

5 Discussion

In this work, we assume that there is a connection between the quality of software and the quality of its development. The process and the tools used in software's development can provide an insight into the role of teams and their involvement in software development and maintenance, and modeling, this enables us to log:

– The evolution of code, with pull requests and recorded issues;
– The evolution of teams and subteams, alongside code versions;
– Reports detailing performance, test results, errors and analytic details.

We now discuss, how this model can be used to support algorithmic accountability, in terms of the principles of accountable algorithms defined in Fairness, Accountability, and Transparency in Machine Learning [1].

5.1 Fairness

Fairness is a core principle, and our PROV model can not ensure that algorithmic decisions do not create discriminatory or unjust impacts when comparing across different demographics. It does not support this definition of fairness because it does not describe how an algorithm works, or it's inputs and outputs. It provides a meta-level of information which can be used alongside other approaches that directly model algorithms and their inputs and outputs.

5.2 Auditability

For an algorithm to be auditable it should enable third parties to probe, understand, and review the behavior an algorithm, and enables monitoring, checking, or criticism. Where appropriate the provision of detailed documentation, APIs, and permissive terms of use. The PROV model supports auditability by providing an accessible model which can be queried using SPARQL. It enables third parties to monitor and check: the processes and protocols used to develop the software; testing strategies; who played a role in the development of a particular software feature; and what events happened during development on bug fixes and new features and, in general, the evolution of a software system.

[1] Principles for Accountable Algorithms and a Social Impact Statement for Algorithms: http://www.fatml.org/resources/principles-for-accountable-algorithms

5.3 Responsibility

To make algorithms responsible there should be externally visible avenues of redress for adverse individual or societal effects, and there should be a designated internal person who is responsible for the timely remedy of any reported issues.

Our model does not support this definition of responsibility, it does not provide an avenue of redress or identify who is responsible for a particular part of the code. However, it does provide a record of who is involved with the development of code features and bug fixes, who reports issues, and the evolution of team members. And while our model can expose who is attributed to a particular part of the code, it should not be used to identify who is responsible for redress; Our approach may only identify coders who are not always responsible for the policies or design decisions which may be put in place by their company or institutions. However, the information in our PROV model may support the person dealing with reported issues.

5.4 Explainability

For an algorithm to be explainable it should ensure that algorithmic decisions, as well as any data driving those decisions, can be explained to end-users and other stakeholders in non-technical terms.

In terms of algorithmic accountability, our PROV model would not be able to speak to algorithmic decisions or the data driving those decisions. However, it can be used to support the explainability of software development. The PROV model provides a formalised model to described sequences of activities and entities, and their relationship to agents, and it is not to most part human understandable. While the PROV data model is simple, it is hard for humans to parse because PROV documents tend to become exceptionally complex when modelling just a few PROV elements. There has been research to improve their understandability, some of this research has been used to convert a PROV model into human readable sentences [6, 5], which could be applied to the response of a SPARQL query.

5.5 Accuracy

Identify, log, and articulate sources of error and uncertainty throughout the algorithm and its data sources so that expected and worst case implications can be understood and inform mitigation procedures.

The PROV model does not directly support this principle, however, by formalising a log of information from multiple sources into a single connected model we can provide insight into the events around an error and uncertainty. Understanding the development process and the events which happened around the time of the error could be used to monitor patterns in development, such as software testing practices, the involvement of specific team members, or rushed deployment of a new software version. Monitoring the patterns of multiple software projects who have reported 'fairness' issues, could reveal the sequence of events or practices that is a symptom of them, and thus could inform software development best practices.

6 Conclusion

The information provided in the APIs of web-based hosting services for version control, and any inbuilt and third-party tools, can be used to create a detailed PROV model describing the software development of algorithms. Their use of RESTful APIs allows this information to be modelled using PROV. These models describe the evolution of software and the teams that developed them, and this software evolution is supported by resolvable reports from inbuilt and third-party tools.

While these models do not solve algorithmic accountability because they describe algorithms from a software development perspective, they can be used to support it. We can describe the events during the development of an algorithm, and thus we can describe and recognise events that may relate or lead to unfairness.

For future work, we plan to develop tools to build provenance models around a project's hosted in web-based services for version control and apply this to various projects. We will analyse these PROV graphs to establish patterns, describe classifications, and identify metrics around fairness and fitness for use.

References

1. De Nies, T., Magliacane, S., Verborgh, R., Coppens, S., Groth, P.T., Mannens, E. and Van de Walle, R. Git2PROV: Exposing Version Control System Content as W3C PROV. In International Semantic Web Conference (Posters & Demos) (pp. 125-128). (2013)
2. Loeliger, J.: Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development. OReilly Media, Inc. (2009)
3. Groth, P., Moreau, L.: PROV-Overview: An Overview of the PROV Family of Documents. W3C Working Group Note (2013)
4. Dabbish, Laura, et al. Social coding in GitHub: transparency and collaboration in an open software repository. Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work. ACM, 2012.
5. Richardson, Darren P., and Luc Moreau. Towards the domain agnostic generation of natural language explanations from provenance graphs for casual users. International Provenance and Annotation Workshop. Springer, Cham, (2016)
6. Packer, Heather S. and Moreau, Luc Generating narratives from provenance relationship chains. At Hypertext and Narrative Hypertext and Narrative, Cyprus. pp. 37-41 (2015)