GitHub2PROV: Provenance for Supporting Software Project Management

Heather S. Packer
University of Southampton
Southampton, UK
hp3@ecs.soton.ac.uk

Adriane Chapman
University of Southampton
Southampton, UK
adriane.chapman@soton.ac.uk

Leslie Carr
University of Southampton
Southampton, UK
lac@soton.ac.uk

Abstract
Software project management is a complex task that requires accurate information and experience to inform the decision-making process. In the real world software project managers rarely have access to perfect information. In order to support them, we propose leveraging information from Version Control Systems and their repositories to support decision-making. In this paper, we propose a PROV model GitHub2PROV, which extends Git2PROV with details about GitHub commits and issues from GitHub repositories. We discuss how this model supports project management decisions in agile development, specifically in terms of Control Schedule Reviews and workload.

CCS Concepts • Theory of computation → Data provenance; • Software and its engineering → Agile software development.

Keywords process provenance data, software project management, PROV

ACM Reference Format:
https://doi.org/10.1145/nnnnnnnnnnnnnn

1 Introduction
Software project management is the application of knowledge, skills, and tools to manage activities to meet project requirements [6]. It requires managers to balance cost, scope, and schedule, to meet these requirements. This balance is difficult to achieve and poor management results in financial losses. The Project Management Institute (PMI) reports that 9.9% of every dollar invested in projects is wasted1, and


Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. TAPP ’19, June 3, 2019, Philadelphia, PA © 2019 Copyright held by the owner/author(s).

around $1 million is wasted every 20 seconds or $2 trillion every year2.

The PMI attributes wastage to three factors: 1) Failing to bridge the gap between strategy design and delivery; 2) Executives do not recognise that their strategy is delivered through projects; and the most relevant to this paper 3) the “importance of project management as the driver of an organization’s strategy isn’t fully realized”3. To enable software project managers to make rational critical decisions, they require complete, unbiased and accurate data. However, in the real world decision makers rarely have access to perfect information, and data may be so poorly presented that a decision is not obvious4. It can be hard for managers to access data because of the opacity of software engineering tools. They usually make decisions under time pressure and with inadequate data of questionable accuracy. Often decisions are based on a manager’s experience and trusted parties’ opinions5.

Version Control Systems (VCS) such as Git, are commonly used for project development. They use branching, the duplication of a commit, for developing new features in parallel along both branches. These VCS repositories are typically hosted on cloud services such as GitHub, because they provide additional tools for development teams to manage and track their projects. Both VCS and hosting services offer flexibility in their use, and thus teams’ working practices can vary greatly especially because there is no standardised process on how to interact with them.

Provenance data models describe the lineage of data and this can describe the software engineering process. The Git2PROV [3] tool already models Git processes using PROV but this model data does not describe collaborations relating to decisions behind commits. Git process handles the versioning of files, whereas GitHub provides all of the social aspects of development, such as issue reporting and development collaboration. The complexity of these social interactions is compounded by the number and types of development branches (see Figure 16), and distributed development. In

2Brightline Initiative: https://www.worldfinance.com/strategy/the-brightline-initiative-gives-life-to-ideas

3PMI Critical Decision Making Skills for Project Managers: https://www.pmi.org/learning/library/critical-decision-making-skills-project-managers-5798

4A successful Git branching model: https://nvie.com/posts/a-successful-git-branching-model/
order for software project managers to track and analyse the development of features they require more social information than Git2PROV can provide to predict the time required to develop features for creating schedules. In order to better support them, we propose GitHub2PROV. It uses PROV to model social interactions influencing software project development, specifically issues. Concretely, in this paper we:

1. Present a PROV model called GitHub2PROV, which extends the Git2PROV model;
2. Show how GitHub2PROV can provide insights into software development from multiple sources, both GitHub API and git file;
3. Evaluate our model against metrics described in related work to support decision-making in agile development.

The rest of the paper is as follows. In Section 2 we describe software development management in terms of Agile development using Git and Git web hosted repositories. We then, in Section 3 describe related work including PROV, Git2PROV, PROV workflows and process provenance. Then in Section 4, we present our model GitHub2PROV. Following that, in Section 5 we discuss how our model can support project managers. In Section 6 we provide a discussion about our model, its uses and limitations. Finally, in Section 7 we conclude and present our future work.

2 Software Development Management

Adaptive life cycles, also known as agile methods, are intended to respond to high levels of change. They are iterative and incremental, and unlike other methods, their iterations are very rapid with iterations usually have a duration of 2 to 4 weeks [6]. Git manages low-level version control transactions. It is the de facto standard for agile software development when it comes to version control systems5. It supports a range of workflows and allows changes to be quickly pushed down the development pipeline.

There are many tools that are used to support Agile software engineering from communication through source repository. We now discuss the popular tools, Git and GitHub, an example of a web hosted repository.

2.1 Git and Web Hosted Repositories

Git is a VCS, where users can create a snapshot of files by committing them to Git. Typically a user would follow this process to create a git commit8:

1. They create files on their local machine;
2. Then they stage these files ready for a commit;
3. The staged files are then pushed to a Git repository.

This Git repository can be hosted either locally or on an online repository hosting service.

Each committed snapshot is stored within a Git repository, allowing users to view changes between commits and revert code to previous versions. A series of commits can be described as a graph, where it is possible to create separate chains of commits, called branches. A repository will usually have a main or Master branch that is considered the primary version of the code. Additional branches allow users to diverge from the Master branch so that work can continue on features without changing the Master. Software projects and developers have different approaches to using branches to manage the evolution of a codebase. For example, Gitflow7 proposes a Master branch to hold production-ready code, and a Develop branch where the code is assembled until it has reached a stable point suitable for merging to Master. Development work on features and bug fixes is conducted on separate branches after which developers will merge to Master.

Web services such as GitHub8 and Bitbucket9, allows users to publish their Git repositories online. Git repositories are commonly hosted on web services because they allow users to collaborate on the development of features, and report

---

5Atlassian Agile Coach: https://www.atlassian.com/agile/software-development/git
6The numbered stages be described using Git2PROV
7https://nvie.com/posts/a-successful-git-branching-model/
8GitHub: http://github.com
9BitBucket: http://bitbucket.com
issues. The way in which users interact with GitHub and similar services can vary greatly due to a team’s working practices and the range of features offered to manage changes; there is no standardised process for interacting with Git.

2.2 Agile Software Engineering and Management Metrics

Software project managers require tools to make sense of project development. There are numerous third party tools available via GitHub’s marketplace, these tools however provide limited customisation on the data provided. The focus of our work is to develop a model that is flexible enough for managers to construct their own queries. These queries can be used to create a Control Schedule to monitor the status of the project [6]. The goal of this process is to identify any deviations from the plan so that corrective and preventive actions can be taken.

A Control Schedule (taken from A Guide to the Project Management Body of Knowledge (PMBOK) [6]) determines:

- **P1** The total amount of work delivered;
- **P2** The rate at which deliverables are produced;
- **P3** Whether deliverables are validated; and
- **P4** The velocity of a team.

3 Background and Related Work

In order to model provenance, we will be using the W3C standard language PROV [4]. PROV describes provenance using a conceptual model PROV-DM, which can be serialized using the PROV formats PROV-O, PROV-XML, PROV-N, and queried using PROV-AQ. It has three core concepts, `prov:Entity`, `prov:Activity`, and `prov:Agent`, and their relationships.

The Git2PROV tool [3] was developed to map a Git repository to the PROV format. The tool generates PROV (see Figure 2) describing:

1. The activity of committing a set of files. This is expressed as a `prov:Activity` that connects two `prov:Entity` objects, expressed through `prov:used` and `prov:wasGeneratedBy` relations. Figure 2 shows PROV of a commit `c` which uses a file `f_{c-1}` and generates a file `f_c` [3];
2. The version of the files that were committed and how they relate to the previous versions. This is represented using a dependency between two objects expressed as the relationship between two `prov:Entity` objects using the relations `prov:wasDerivedFrom` and `prov:specializationOf`, where a file `f_c` was derived from another previous file `f_{c-1}`, both are a specialization of a certain file `f` [3];
3. Agents who authored and committed the files, are expressed as `prov:Agent` which created the Entity using the relations `prov:wasAttributedTo` and `prov:wasAssociatedWith`, modeling the two potentially distinct roles of a author and a committer [3].

While Git2PROV describes the chain of commits within a Git repository, on both locally and cloud-hosted services, it does not describe the collaborations that those cloud-hosted services facilitate.

In addition to the Git2PROV, the provenance community has also explored PROV to document the Software Development Process (SDP), this research area was identified in 2010 as an emerging field in [9] and it has not yet filtered through into software project management. The following related work discusses provenance and SDP.

Miles et al. [8] propose PRiME, a technique that adapts application designs to enable them to interact with a provenance middleware layer. They detail how to apply PRiME and analyse its effectiveness using two case studies. Wendel et al. [11] present a solution to common failures in software development processes based on PRiME. Notably, they identify that users are concerned with quality reassurance, process validation, statistical analysis, and process optimisation. Junaid et al. [5] propose using a system, called iSPuP, that intercepts users actions with suggestions of possible future tasks, based on captured provenance. iSPuP aims to improve processes and their focus is to support software process execution, monitoring, and analysis phases to improve software processes.

Costa et al.’s [2] work investigates whether process provenance data can provide information to support software process execution, monitoring, and analysis phases, contributing to the process improvement as a whole. Notably, their paper conducted a survey of approaches describing software process execution and found 16 publications. These papers either suggested the use of generic tools for process management or adopt propriety solutions. None of these proposals uses data provenance.

Costa et al. [1] presents an approach for visualising software process provenance data, called SPPV. To evaluate their work they run an exploratory analysis using two real-world industry processes, with positive feedback from the process
experts. In their evaluation they investigated whether prove-
nance data could support managers in identifying problems
with workloads, using the following questions [1]:

Q1 How many entities/artifacts were manipulated by the agent?
Q2 How often are these entities manipulated by the agent over
time?
Q3 How many activities/tasks were completed by the agent?
Q4 How often are these activities performed by the agent over
time?
Q5 How many activities/tasks, related to new requests, were
started by the client?

4 Mapping of GitHub to PROV

In order to investigate whether PROV models can inform
software project management, we use the GitHub’s API[10].
We selected GitHub because it has become a de facto standard
for software development, with over 31 million developers
and 96 million repositories in 2018[11], and its API is RESTful
which allows resources to be modeled as resolvable elements
in a PROV graph. Our model, GitHub2PROV, makes use of
two core resources within a GitHub project, Commits, and
Issues. We describe the mapping of these two resources in
the following two sections.

4.1 Commits

In this paper, we discuss Commit resources which are re-
ferred to by both Git2PROV and our GitHub2PROV model.
We will use the naming convention Git commit and GitHub
commit to describe commits provided by Git2PROV and
GitHub2PROV models. While we make the distinction be-
tween Git commit and GitHub commit, these two activities de-
scribe the same commit activity. Git2Prov’s Git commit is not
resolvable, hence we utilise the URL provided by GitHub’s
API to describe the GitHub commit entity. Concretely, resolv-
ing a GitHub commit entity’s URL provides details about the
committer, author, message, and verification.

GitHub’s API describes a GitHub commit resource and
provides details including the GitHub commit’s message,
any comments, build status, the author and committer. Our
mapping is shown in Figure 3, and the two classes that we
used are identified below. For each class, we describe how
provenance can be expressed using concepts from the PROV
Data Model [10]:

1. Communication - where github_commit is informed
   by github_parent_commit_1 and github_parent_com-
   mit_2. Where github_parent_commit_2 is optional. Hav-
   ing two parent GitHub commits represents a merge.
2. Attribution - where the github_commit activity is at-
   tributed to github_committer using the prov:wasAss-
   ociatedWith relation.

In order to map our model onto Git2PROV, we use a com-
unication relation, wasInformedBy, between a Git commit
and GitHub commit (see Figure 3).

4.2 Issues

GitHub has an issue tracking system which manages and
maintains lists of issues. It is useful in Agile development
because they support collaboration, particularly in large or
geofraphically distributed communities. GitHub’s API de-
scribes an issue resource and details a list of events that have
occurred against that issue. In order to model issues, we have
identified three cases:

1. A new issue (see Figure 4), where an issue has been
   created in a project. We describe our model using con-
   cepts from the PROV Data Model [10]:
   a. Communication - where annotator_commit is in-
      formed by issue_event_annotation.
   b. Attribution - where github_annotator was attrib-
      uted to issue_version_annotation.
   c. Dependency - issue issue_version is a specialization
      of a certain issue issue.
   d. Activities - an event issue_event generates a new
      issue version issue_version;

Figure 3. Github2PROV and Git2PROV commit, attributes for type, state and label not shown for clarity.
5 Applying the GitHub2PROV Model

In order to evaluate GitHub2PROV, we discuss how it can support (i) the compilation of a Control Schedule Review (see Section 2.2), and (ii) workload (see Section 3), using metrics derived from the GitHub2PROV. We note that the metrics we discuss in this Section are useful in project management, however, none of them provide insight into the causes of the results but they do provide discussion points to evaluate a team’s progress.

In order to evaluate our metrics, we have generated GitHub2PROV and GitHub2PROV models for the LibVips project, a popular imaging processing library which was first developed in 1995 and its Git repository was first pushed to GitHub in 2007. At the time of generating the model, it had 5,684 GitHub commits, 51 open issues, and 998 closed issues. In Figure 6 we depict an excerpt of the LibVips provenance.

5.1 Control Schedule Review

In order to discuss how our model can support a Control Schedule Review, we first identify metrics that can be used (see Section 2.2):

- **M1 Open/close rates** - how many issues are reported and how many issues closed within a specific time period. This metric describes the volume of work that has been completed and supports P1.
- **M2 Lead Time** - how long it takes for a feature to be in the delivered software. This metric supports P2, which can be used to provide context on the rate at which deliverable is produced on the project.
- **M3 Cycle Time** - how long it takes for a change to be made to the software system and deliver that change into production. This metric helps describe the rate at which deliverable is produced, thus supporting P2.
- **M4 Team velocity** - how many “units” of software the team typically completes in an iteration. This supports P4.

The above metrics do not address item P3, this is because our model, for now, does not describe validation processes. We plan to address this in future work.

A common requirement in the evaluation of metrics used for a Control Schedule Review is the start and end time for a given activity such as the development of a feature. The use of our model allows the explicit timing information derived from a traditional issue tracker to be combined with the implicit timing provided by other activities such as commits. The activity start is defined as the earliest activity directly attributable to the issue. The activity end would usually be the point in time that code is merged to production, after which CI/CD pipelines or code release processes take over.

In Table 1, we present a description of our metrics, which PROV elements are used to calculate the metrics, and metric values taken from the month of January 2019 from the

### Table 1

<table>
<thead>
<tr>
<th>PROV element</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue Entity</td>
<td>prov:type</td>
<td>title of issue state</td>
</tr>
<tr>
<td>Issue Event Activities</td>
<td>prov:type</td>
<td>title of issue open/closed</td>
</tr>
<tr>
<td></td>
<td>endTime</td>
<td>title of issue xsd:dateTime</td>
</tr>
</tbody>
</table>


LibVips exemplar. We have omitted for Team Velocity M4 from the table because we were unable to calculate it from the LibVips PROV model. To calculate M4 we would require each issue to be annotated with the number of software development "units", which is usually estimated by one or more team members. We queried the models using SPARQL, and the following is an example query for open rates (M1):

```sparql
SELECT ?i ?st
WHERE {
  ?i a github2prov:issue.
  ?i prov:wasGeneratedBy ?ca.
  ?ca prov:startedAtTime ?st.
  FILTER (( ?st >= "2019-01-01T00:00:00+00:00"^^xsd:dateTime) &&
           ( ?st < "2019-02-01T00:00:00+00:00"^^xsd:dateTime))
}
```

5.2 Workload

In Section 3 we describe metrics used to determine the workload of users in project development scenarios. These metrics could be used to support the allocation of work during a Control Schedule Review. To make the metrics more directly applicable to our model, we defined them in terms of...
The number of files and commits (described by Git2PROV), issues and issue events (described by our GitHub2PROV model). These metrics are presented in Table 2, this table also defines the PROV elements used, and metric values taken from the month of January 2019 from the LibVips exemplar using SPARQL.

### Table 1. Control Schedule Review Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
<th>Features used in PROV</th>
<th>LibVips Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>Open/Closed Rates</td>
<td>GitHub2PROV: issue_event activity startTime attribute, issue_event activity status attribute</td>
<td>Opened: 21, Closed: 23</td>
</tr>
<tr>
<td>M2</td>
<td>Lead Time</td>
<td>GitHub2PROV: issue_event activity startTime attribute</td>
<td>Days: 22.78</td>
</tr>
<tr>
<td>M3</td>
<td>Cycle Time</td>
<td>GitHub2PROV: issue_event activity startTime attribute</td>
<td>Days: 20.71</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
<th>Features used in PROV</th>
<th>LibVips Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>M5</td>
<td>The number of files that were edited by an agent (Q1)</td>
<td>Git2PROV: file entities, committer agents</td>
<td>Developer 1: 122, Developer 2: 7, Developer 3: 8, Developer 4: 2, Developer 5: 2</td>
</tr>
<tr>
<td>M6</td>
<td>The frequency of files which are manipulated by an agent (Q2)</td>
<td>Git2PROV: file entities, committer agents</td>
<td>Developer 1: 99.34, Developer 2: 0.36, Developer 3: 1.02, Developer 4: 1.00, Developer 5: 1.00</td>
</tr>
<tr>
<td>M7</td>
<td>The number of events an agent is associated with (Q5)</td>
<td>GitHub2PROV: issue_event activity, annotator/opener agents</td>
<td>Developer 1: 86, Developer 2: 3, Developer 3: 27, Developer 4: 5, Developer 5: 10</td>
</tr>
<tr>
<td>M8</td>
<td>The frequency of events associated with an agent (Q4)</td>
<td>GitHub2PROV: issue_event activity, commit activity, annotator/opener/committer agents</td>
<td>Developer 1: 38.30, Developer 2: 3.14, Developer 3: 4.63, Developer 4: 4.00, Developer 5: 4.00</td>
</tr>
<tr>
<td>M9</td>
<td>The number of events relating to new issues that an agent is associated with (Q5)</td>
<td>GitHub2PROV: issue_event activity, commit activity, annotator/opener agents</td>
<td>Developer 1: 71, Developer 2: 1, Developer 3: 11, Developer 4: 4, Developer 5: 0</td>
</tr>
</tbody>
</table>

### Table 2. Workload Metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
<th>Features used in PROV</th>
<th>LibVips Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>Open/Closed Rates</td>
<td>GitHub2PROV: issue_event activity startTime attribute, issue_event activity status attribute</td>
<td>Opened: 21, Closed: 23</td>
</tr>
<tr>
<td>M2</td>
<td>Lead Time</td>
<td>GitHub2PROV: issue_event activity startTime attribute</td>
<td>Days: 22.78</td>
</tr>
<tr>
<td>M3</td>
<td>Cycle Time</td>
<td>GitHub2PROV: issue_event activity startTime attribute</td>
<td>Days: 20.71</td>
</tr>
</tbody>
</table>

6 Discussion

Our approach uses the GitHub API to populate the GitHub2PROV model. Thus it contains the same data as the GitHub API, however, our model benefits from the lineage of data provided by the PROV-DM, which was used to support queries M2 and M3. Another alternative to querying our model using SPARQL, is to place all statements into a relational database, however 2 out of our 8 use cases requires graph traversal and using a database would be harder in terms of lines of code and query complexity.

Our model describes GitHub commits and issues, both of which are core to project development. It does not model every possible interaction with GitHub, we took this design choice for two reasons: First, we wanted to evaluate whether the information about GitHub commits and issue tracking would be useful, without investing in modeling GitHub’s extensive API; Second, our model describes core concepts that are common to online Git hosting services so that our model could be re-purposed for other services.

It is possible to extend GitHub2PROV so that it describes other resources made available via the GitHub API, however it is a challenge to capture the complete provenance of this process because project development is often distributed over a number of non-integrated tools. Provenance capture may not be complete, thus making it difficult to draw conclusions from these sources or it may be extremely time-consuming to collect due to the scale of available data.

The results from the metrics can be displayed in tables or graphs so that software project managers can quickly assess the metrics (see Figure 7). The metrics derived from the project management perspective (see Section 5.1) required more complex queries than the metrics derived from related provenance research (see Section 5.2). The former metrics were more complex because it was necessary to traverse the directed graph of Git commits to determine when development on an issue started and this required insight into a project’s development process. While we have calculated measures for the LibVips repository, where we have generated our figures for the time period of a month, this may not be representative of development iterations. This knowledge should, however, be available to a project manager using this model. Even without the knowledge of these development details, our model can still provide an insight into the activity of a project, the evolution of a team, and the resolution of issues. This insight can provide users with details about whether the code base is maintained and whether the developers are reactive to issues. It also allows users to compare GitHub code libraries for use in their projects.

We were also unable to generate a metric M4 from our exemplar, team velocity, because the developers did not provide information in the repository about development units. If the developers choose to include this level information in their issues, this could be calculated. The richness of PROV
captured by our model depends on how developers interact with GitHub. For instance, if the development is led by a single developer, and they do not report issues through the issue tracker then this information is lost. In this case, a better strategy would be to use the time derived from tasks completed tasks for project managing workload.

While our model extends the Git2PROV model, it is not required to query a GitHub2PROV model. That said, Git2PROV provides interesting insights into Git repositories. The combination of Git2PROV models of multiple user repositories and GitHub2PROV model provide interesting insights into team behaviour. We also note that GitHub2PROV would not be able to support the Metrics M5 and M7 without the Git2PROV model. Other provenance models could be combined with the GitHub2Prov model to enrich it and more applicable for other purposes. For instance, the work of Miao et al. [7] presents a model for data scientists to model the evolution of their datasets, the dataset version could be connected to a GitHub commit.

7 Conclusions and Future Work

In this paper, we present GitHub2PROV, a PROV model which describes GitHub commits and Issue resources. It extends the Git2PROV model. We discuss how our model can support project management decisions using metrics and we calculated these metrics for our LibVips exemplar.

For future work, we plan to extend our model by including other core concepts from the GitHub API namely pull requests, other user interactions, and information from continuous integration tools describing validation. By including these concepts in our model, we can provide better support to software project managers. We plan to explore addition use cases for our model, including Software Sustainability for open source projects.

References