Extracting Artifact Lifecycle Models from Metadata History

Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo, Canada
{obaysal, okononen, rtholmes, migod}@cs.uwaterloo.ca

Abstract—**Software developers and managers make decisions based on the understanding they have of their software systems. This understanding is both built up experientially and through investigating various software development artifacts. While artifacts can be investigated individually, being able to summarize characteristics about a set of development artifacts can be useful. In this paper we propose *lifecycle models* as an effective way to gain an understanding of certain development artifacts. Lifecycle models capture the dynamic nature of how various development artifacts change over time in a graphical form that can be easily understood and communicated. Lifecycle models enables reasoning of the underlying processes and dynamics of the artifacts being analyzed. In this paper we describe how lifecycle models can be generated and demonstrate how they can be applied to the code review process of a development project.**

I. Problem

Software development projects often face challenges when trying to leverage development data that accrues in various repositories (e.g., issue tracking systems, version control systems, etc.) during software development. The volume of data in these repositories makes it difficult to manually generate a reasonable understanding of both the state of the data and how it has evolved over time. In this paper we introduce a data pattern called lifecycle models that can be used to extract a succinct summary of how specific properties within a development artifact have changed over time.

Lifecycle models can be applied to any data that changes its state, or is annotated with new data, over time. For example, issues often start their lives as `OPEN` and are eventually `RESOLVED` or marked `WONTFIX`. A lifecycle model for issues could build up an aggregate graphical representation of how bugs flow through these three states. For example, the lifecycle model would capture the proportion of bugs that are reopened from a `WONTFIX` state that might be interesting for a manager considering changes to their issue triage process. Such lifecycle models often exist in process models (if defined). Therefore, extracting one from the history enables comparing the defined lifecycle with the actual one.

In this paper we will apply lifecycle models to instead capture how the code review process works in both Mozilla and Webkit. We demonstrate how these models both capture interesting traits within individual projects, but are also succinct enough to compare traits between projects.

II. Solution: Artifact Lifecycle Models

Lifecycle models can be extracted for any data elements that evolve over time from their metadata history. These elements may include issues' change status, reviewed patches, evolved lines of code. By examining how each artifact evolves over time, we can build a summary that captures common dynamic evolutionary patterns. Each node in a lifecycle model represents a state that can be derived by examining the evolution of an artifact.

The lifecycle model is presented as a simple graph that shows the states the model contains; edges capture the transitions between the lifecycle states. An example lifecycle model is given in Figure 1.

Lifecycle models can be generated as follows:

1) Determine key states of the system or process (e.g., a number of events that could occur) and their attributes.
2) Define necessary transitions between the states and specify an attribute for each transition.
3) Define terminal outcomes (optional).
4) Define and gather qualitative or quantitative measurements for each state transition and if present, final outcomes. In addition to these measurements, the time spent in the state or the transition to another state can also be considered and analyzed.

The pattern provides means to model, organize, and reason about the data or underlying processes otherwise hidden in individual artifacts. While lifecycle model can be modified or extended depending on the needs, they work best when the state space is well defined. Applying this pattern to complex data may require abstraction.

III. Example: Code Review Lifecycle Models

In this section we apply a lifecycle model to the code review process to try to highlight how code reviews happen in practice. The goal of the model in this instance is to identify the way that patches typically flow through the code review process (and identify exceptional review paths). We have extracted the code review lifecycle model for Mozilla Firefox; all events have been extracted from Mozilla's public Bugzilla instance. A complete description of this study has been previously published [1].

All code committed to Mozilla Firefox undergoes code review. Developers submit a patch containing their change to

978-1-4673-6296-2/13 © 2013 IEEE
DAPSE 2013, San Francisco, CA, USA

Accepted for publication by IEEE. © 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

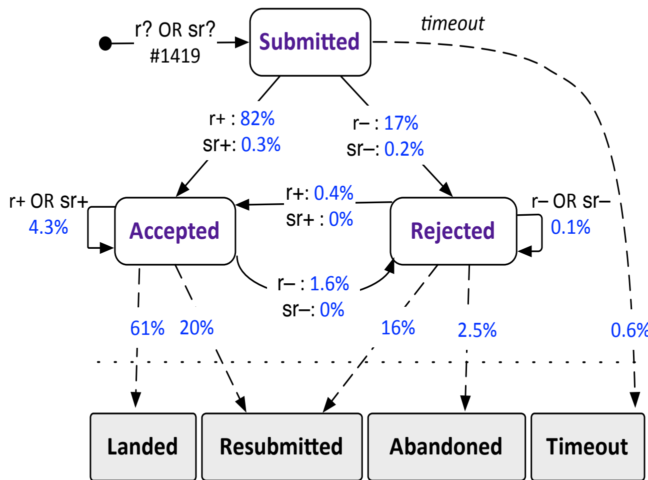

Fig. 1. Patch lifecycle of the Firefox core contributors.

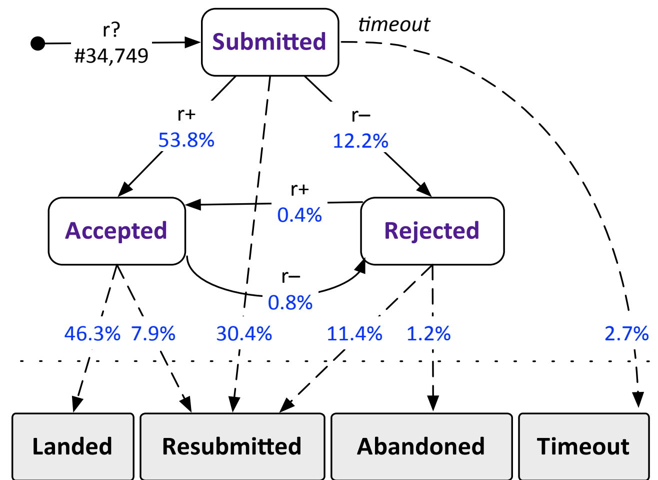

Fig. 2. Patch lifecycle of the WebKit project.

Bugzilla and request a review. Reviewers annotate the patch either positively or negatively reflecting their opinion of the code under review. For highly-impactful patches super-reviews may be requested and performed. Once the reviewers approve a patch it can be applied to the Mozilla source code repository.

We generated the model by applying the four steps mentioned previously:

1) **State identification:** Patches exist in one of three key states: `submitted`, `accepted`, and `rejected`.
2) **Transition extraction:** Code reviews transition patches between the three primary states. A review request is annotated with $r?$. Positive reviews are denoted with a $r+$. Negative reviews are annotated $r-$. Super reviews are prepended with an s (e.g., $sr+$/$sr-$).
3) **Terminal state determination:** Patches can also exist in four terminal states which are defined considering entire patch history to resolve an issue. `Landed` patches are those that pass the code review process and are included in the version control system. `Resubmitted` patches are patches that a developer decides to further refine based on reviewer feedback. `Abandoned` patches capture those patches the developer opted to abandon based on the feedback they received. Finally, the `Timeout` state represents patches for which no review was given even though it was requested.
4) **Measurement:** For this study we measured both the number of times each transition happened along with the median time taken to transition between states.

Figure 1 illustrates the lifecycle model of the patch review process for core Mozilla contributors. The lifecycle demonstrates some interesting transitions that might not otherwise be obvious. For instance, a large proportion of accepted patches are still resubmitted by authors for revision. Also, we can see that rejected patches are usually resubmitted, easing concerns that rejecting a borderline patch could cause it to be abandoned. We also see that very few patches timeout in practice. From the timing data (not shown for clarity) we see that it takes an average of 8.9 hours to get an initial $r+$ review, while getting a negative $r-$ review takes 11.8 hours. Application of this pattern on the code review process of Firefox (and comparing the lifecycle models for patch review between core and casual contributors) has generated a discussion and raised concerns [2] among the developers on the Mozilla Development Planning team:

> "... rapid release has made life harder and more discouraging for the next generation of would-be Mozilla hackers. That's not good... So the quality of patches from casual contributors is only slightly lower (it's not that all their patches are rubbish), but the amount of patches that end up abandoned is still more than 3x higher. :-(" [Gerv Markham]

> "I do agree that it's worth looking into the "abandoned" data set more carefully to see what happened to those patches, of course." [Boris Zbarsky]

The lifecycle model can be easily modified according to the dataset at hand. For example, we have recently applied the pattern to study the code review process of the WebKit project. The model of the patch lifecycle is shown in Figure 2. While states remain the same, there are fewer state transitions since the WebKit project does not employ a 'super review' policy. Also, the self edges on the accepted and rejected states are absent because while Mozilla patches are often reviewed by two people, Webkit patches only receive individual reviews. Finally, a new edge is introduced between `Submitted` and `Resubmitted` as Webkit developers frequently 'obsolete' their own patches and submit updates before the receive any reviews at all.

Comparing Figure 1 and Figure 2, similarities and differences between the two code review processes become apparent. For example, the ratio of $r+ : r-$ on initial submissions is about the same, and both lifecycles show similar rates of

transition between the Accepted and Rejected states. In contrast, WebKit patches are more likely to be resubmitted (50% vs. 36% for Firefox), among which 30.4% are resubmitted without receiving a feedback. In terms of patch review times we can see that Firefox patches are on average accepted three times faster (8.9 hours vs. 30 hours) and rejected six times faster (11.8 hours vs. 80 hours) than WebKit patches.

IV. Other Applications

The lifecycle model is a flexible means that can be used in a variety of software investigation tasks. For example:

- Issues: As developers work on issues their state changes. Common states here would include NEW, ASSIGNED, WONTFIX, CLOSED, although these may vary from project to project.
- Component/Priority assignments: Issues are often assigned to specific code components (and are given priority assignments). As an issue is triaged and worked upon these assignments can change.
- Source code: The evolutionary history of any line or block of source code can be considered capturing Addition, Deletion, and Modification. This data can be aggregated at the line, block, method, or class level.
- Discussions: Online discussions, e.g., those on StackOverflow, can have status of CLOSED, UNANSWERED, REOPENED, DUPLICATE, PROTECTED.

V. Conclusion

In this paper we have introduced the lifecycle model data pattern. This pattern can be used to capture both common and exceptional cases in the evolution of software artifacts. We have applied it successfully to code review in both Mozilla and Webkit and have received positive feedback from Mozilla developers as they investigated their own code review processes. Lifecycle models are easy to generate and interpret enabling them to be used for a wide variety of data modelling applications.

References

[1] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The Secret Life of Patches: A Firefox Case Study," in *Proc. of the 19th Working Conference on Reverse Engineering*, 2012, pp. 447–455.

[2] Mozilla, "The Mozilla Development Planning Forum." [Online]. Available: https://groups.google.com/forum/#!topic/mozilla.dev.planning/6GGuKd6ElEo