Modeling cognitive workload in open-source communities via simulation

Alexey Tregubov, Jeremy Abramson, Christophe Hauser, Alefiya Hussain, and Jim Blythe

USC Information Sciences Institute, Marina del Rey, CA (USA)
{tregubov, abramson, hauser, hussain, blythe}@isi.edu

Abstract. Large open-source projects such as the Linux kernel provide a unique opportunity to analyze many of the socio-technical processes of open-source software development. Understanding how cognitive workload affects the quality of code and productivity of work in such environments can help better protect open-source projects from potential vulnerabilities and better utilize limited developer resources.
In this paper, we present two agent-based simulation models of developer interactions on the Linux kernel mailing list (LKML). We also develop several non-simulation machine learning (ML) models predicting patch reversal, to compare with our agent-based simulation models. In our experiments, simulation models perform better in predicting the expected number of reverted patches than ML models. Results are further improved using an explicit process model within the simulation, modeling the patch view process and associated cognitive load on LKML reviewers when new code changes are introduced by developers. We find that the process model can capture the repeated, structured multi-agent activities within a socio-technical community.

Keywords: Agent-based simulation · cognitive workload simulation · open-source development · Linux kernel mailing list.

1 Introduction

Socio-technical systems, which are affected by both the social interactions of actors within those systems and their technical activities or components, are fundamental to many activities [7]. Agent-based simulations of social media have been successful in modeling and predicting human online activity from observable data in purely social network settings, such as Twitter, Facebook and Reddit. Here we extend such work to model activity in a socio-technical system for which observable data about both social media interaction and the technical operation of the system are available, namely the development of the open-source Linux kernel, combining code-related activity in the Git repository with a public mailing list for discussion. The Linux kernel is a very wide-reaching software artifact, embedded in many small IOT devices as well as most of the servers in cloud systems. Developers from across the world contribute to its high quality open-source code framework in a collaborative structure that combines self-organizing and top-down aspects.

In order to model agent behaviors in this context that can be customized to observation data, we make use of a multi-agent process that is repeated many times within the community, and within which agents play various roles. For example, a software development simulation may include many instances of a process in which agents propose software patches, other agents review and comment on them, and another agent makes the decision whether to include the proposed patch. These processes are enacted by human agents rather than machines, and are informal: steps may be skipped or reordered, and process instances may cease before completion. However the set of unfolding process instances still provides a framework to predict the agents' future actions and need for resources.

We present a case study of modeling the software development process in the Linux kernel using the process model shown in Fig 2, and learning the parameters of both agents and processes from publicly available data on the discussion and commit activities of many thousands of individuals. We show that our process-enhanced simulation outperforms an agent simulation without the process model, which in turn often outperforms a non-simulation approach for predicting the number of errors found in software patches based on activity six months earlier in the sub-community in which the patch is proposed. We hypothesize that the agent-based systems can predict which agents will have a higher cognitive load in the later time period based on development and reviewing demands evident in the earlier time period. Higher cognitive load leads both to a greater number of mistakes by experienced performers and also to an influx of less experienced performers, with higher error rates, to meet the increased demand. The agent model naturally captures the variation in response to cognitive load among different agents, and the diffusion of cognitive load through sub-communities within the development community. The simulation that makes use of process models further captures the timeline of increased cognitive load.

This paper makes two main contributions. First, we present an agent-based model of open-source software development, including the recruitment of developers and code reviewers and on-going communication about software under development. Second, we demonstrate the value of a process model that captures repeated, structured multi-agent activities within a socio-technical community of study. We show that our multi-agent simulation improves in prediction of buggy code (in a form of the expected number of reverted patches) over statistical models, and that the predictions are further improved by the use of the process model within the simulation.

2 Related work

There has been relatively little work in agent-based modeling of open-source software development, despite its increasing importance to our modern infrastructure. Spasic and Onggo [12] describe an agent-based model of the software development process within the company AVL which is calibrated using project duration data. Since it models a single company, it does not cover the social protocols of open-source software, and the data used for validation does not cover daily communication, but only project-level features.

Yujuan et al. [9] analyzed 130 developer mailing lists and studied distributed reviewing and integration processes in the Linux kernel. They observe that patches created by more experienced developers are accepted faster. They also found that only 33% of patches make it into the repository and it takes them 3-6 months. We observed similar behavior and used 6-12 months time intervals for simulation test periods.

Blythe et al. [5] describe an approach to model agent behavior on GitHub using the Dash platform [6, 3]. The work described here also uses the Dash platform, but models software discussion on the Linux kernel mailing list in addition to activity on GitHub.

ABMs have been used to model socio-technical interactions in many other fields [7], and they frequently include repeated processes of the kind we investigate here. For example, Kovanis et al. [10] describe an ABM model of peer review process of scientific publications. Outputs of their simulation model fit observational data, which is also the case for simulation models presented in this paper. However, these models do not use an explicit representation of the repeated processes, instead implementing them through agent and environment variables distributed in the simulation. Our use of a general Petri net representation for the process model allows alternative models to be defined and tested, using a general mechanism to integrate them in our simulation architecture and to calibrate the models.

3 Open-source software development

Large open-source software projects such as Linux kernel are developed by hundreds and sometimes thousands of developers, which is why collaboration among them is an essential part of the development process. The most common communication channels, used by open-source software developers, are Git repositories, mailing lists, wiki knowledge repositories, bug and issue tracking systems. Git repositories maintain a project code base as well as a history of changes and comments on code updates from developers. Mailing lists, wikis and issue tracking systems are used for discussions for the upcoming updates and design of the system.

Usually developers propose new code changes to Git repositories by following either formal or informal processes, where new code commits are first proposed and reviewed by peers and then pushed to the main branch of the code. For example, GitHub uses its pull requests and associated review/comment code features. In the LKML new code patches are discussed via emails, and new commits are pushed to the repository when a maintainer, responsible for associated Kernel modules, approves them.

Large projects like the Linux kernel [1], where the development has been done for several decades, accumulate large amounts of historical data on developers' activity. This historical data includes both habits and preferences of individual developers and entire sub-communities within a project (e.g. developer groups of various drivers in the Linux kernel).

The LKML contains artifacts of more than 25 years of communication with more than 4 million messages [2]. Developers organize themselves in groups led by maintainers, making final code approvals before they make it into the main branch of the repository. There are more than 2.5K maintainer groups in the LKML [4]. Often maintainers run their own repositories from which accepted patches make it into the main repository. The Linux kernel has hundreds of active maintainer repositories.

Combined with the hierarchy of Git repositories, the LKML provides a rich source of information on how socio-technical systems function. This information captures many formal and informal socio-technical processes (e.g. new Linux patch development). In this paper, we focus on the Linux kernel patch review process.

4 Simulation

4.1 Framework

We developed our simulation model using the DASH framework [6, 5]. The DASH framework was designed to support agent-based simulations of social networks of different scale and fidelity. It has been used to simulate various social media platforms (e.g. Twitter, YouTube, Reddit, Telegram). It is implemented in Python and supports single-host or cluster modes.

In this project we developed an agent-based simulation of developers, code reviewers and other decision-makers working on the Linux kernel. These workers contribute to a number of GitHub repositories, ultimately feeding into the main Linux kernel repository maintained by Linus Torvalds. They communicate predominantly through the Linux kernel mailing list (LKML), as well as more focused mailing lists and other communications that are not publically available. The agents in our simulation represent LKML users (developers, maintainers, reviewers, etc.). Each agent is characterized by a set of features describing their past activities. For example, how often they participate in patch proposals and discussions via LKML, the patches involved, their associated interest groups and files, and the roles that the agents play in the patch-review process. Agents interact with each other via LKML messages in the simulated environment. Each LKML message includes meta-data about authors, date, patch information, related files and maintainer groups.

The simulation uses a discrete event mechanism, where the event queue schedules the next activation time of each agent. Agents can activate other agents (e.g. by assigning an action like 'review patch') or use self-activation (check in after some time to see new activity in the LKML) Fig. 1 a). When agents are activated they follow the decision process described in Fig. 1 b). Agents produce the following types of LKML messages: 'patch proposal', 'patch comment', 'patch review', 'patch rejection', 'patch commit', 'bug report'.

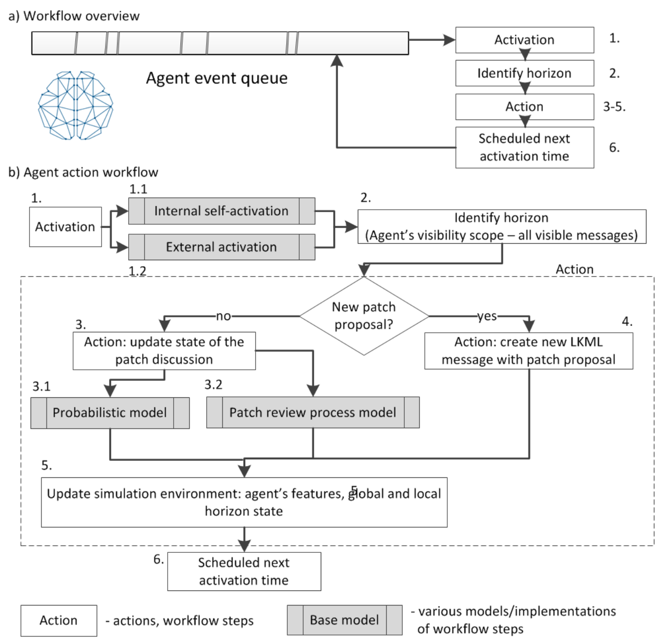
a) Workflow overview
Agent event queue
Activation
1.
Identify horizon
2.
Action
3-5.
Scheduled next activation time
6.
b) Agent action workflow
1.
Activation
1.1
Internal self-activation
External activation
1.2
2.
Identify horizon (Agent's visibility scope – all visible messages)
Action
New patch proposal?
no
yes
3.
Action: update state of the patch discussion
4.
Action: create new LKML message with patch proposal
3.1
Probabilistic model
3.2
Patch review process model
5.
Update simulation environment: agent's features, global and local horizon state
6.
Scheduled next activation time
Action
- actions, workflow steps
Base model
- various models/implementations of workflow steps

Fig. 1. a) Simulation workflow overview, b) Agent action workflow.

4.2 Models

The following broad process is repeated many times in a software development context: (1) an external call or bug-reporting system issues a request for a software patch, to provide new functionality, or fix a bug etc. causes more developers to propose a software patch intended to address the call. (2) Maintainer requests review/feedback on new patches (3) One or more reviewers make a judgment about the proposed patch and make recommendations, either that the patch be accepted into the body of code, or that certain other changes must be made before the patch is re-submitted. (4) A maintainer makes the decision whether or not to accept the reviewed patch into the system. Agents may play different roles in different instances of the same process. For example, an individual may be a developer of software patches in one instance and a reviewer in another. This patch-review process is formalized in Fig. 2.

We developed our process description while examining the software process for the Linux kernel - other software platforms may use slightly different processes. This is an approximate process, followed by humans rather than carried out by machines: the patch request step is often skipped, the review step is skipped less often, and often the request step is unclear. Patches may later be removed from the repository as part of a separate process not modeled in detail here.

We compared two simulation models: (1) probabilistic model, (2) process model. In the probabilistic model agents choose their actions using probability distributions from training data.

In the process model we implemented the patch-review process explicitly. Unlike the probabilistic model, the process model explicitly models patch states and transitions (e.g. 'Revise', 'Do not revise', 'Do not commit') that do not result in visible actions (LKML messages). The simulation maintains a queue of patches in different states (proposed patches, patches with assigned reviewers, patches with assigned revisions, approved by reviewers patches). When new patches are proposed, the simulation creates an instance of the process and triggers an associated maintainer group agent to activate and assign a reviewer or several reviewers to this patch.

In the results section of this paper we compare how these models perform relative to each other and ML models.

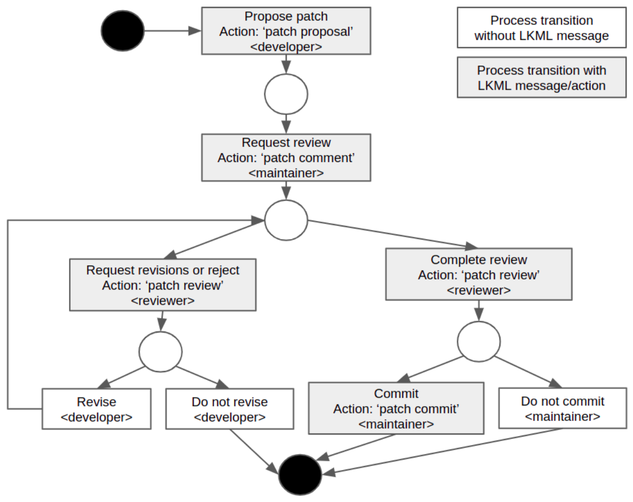

Fig. 2. Patch-review process in LKML.

4.3 Cognitive load model

We compare the ability of our modeling approaches to predict the overall quality of a set of patches, using the number of patches that are reverted as a measure of quality since we are not able to discern patch quality directly. Patches may be reverted for different reasons, usually due to bugs that were later discovered. Patch quality is directly associated with the experience of both the developers and reviewers. The availability of experienced reviewers and how much time they can dedicate for patch reviews also depends on how busy they are. In our description below and in our model, we focus for simplicity on the effect of cognitive load on reviewers, although we would expect to see similar effects on developers and maintainers.

As stated earlier, our hypothesis is that a higher workload (measured by the number of new patches to review) causes a drop in patch quality and hence more reverted patches. When a maintainer group is experiencing a higher workload, reviewers are likely to spend less time per patch and may therefore fail to spot defects. As shown in Figure 4(b), the ground truth data support the hypothesis that the proportion of patch reversions increases as workload increases, measured by the number of patch proposals awaiting reviewers. Our data also show that a higher workload causes relatively inexperienced reviewers to take on more reviews, also leading to a drop in quality.

In our simulation models, the probability of patch reversal is dependent on likelihood of the developer to propose a patch that would be reverted (based on training data statistics) and of the reviewer to miss bugs in the review process. We estimate the probability of patch being reversed as follows: $P = M_{load} \times P_{dev}$, where M_{load} is the reviewer workload/multitasking factor and P_{dev} the developer's probability of submitting a patch that will be reverted. P_{dev} is computed individually for each developer. If there is not enough historical data to compute P_{dev}, we use P_{gr} - historical average probability of patch reversal for the maintainer group to which the patch was proposed.

$$P = M_{load} \times P_{dev} \tag{1}$$

$$P_{dev} = \frac{n\_patches\_reverted}{n\_patches\_proposed} \tag{2}$$

M_{load} is a reviewer workload/multitasking factor, which is a heuristic function that associates length of the work queue with increase of probability to miss a bug in the review process. We used a variation of Weinberg's heuristic ([14, 13]): for nominal workload (a work queue length equal to the historical average for the maintainer group) it is 1.0, if workload doubles (2× of nominal length) then it is 1.2, and if workload triples (3× of nominal length or above) then it is 1.4.

5 ML models for patch reversal prediction

We developed several ML models using different algorithms to predict patch reversal. We translated the patch reversal prediction into a binary classification problem for a given feature vector of a patch discussion (patch features).

We used two groups of features: code features and social features (Table 5). Code features were extracted from patch diff and social features were extracted from LKLM messages. We used a fine-tuned BERT model [8] to calculate text embedding of the original LKML messages stripped from metadata and code. The BERT embedding was also used to calculate keywords of each LKML message. Both BERT embedding and keywords then were used to cluster LKML messages using K-means clustering. Some of these clusters were used in our features. For example, the number of messages in patch discussion in cluster 2. This cluster is characterized by the following top keywords: 'core', 'test', 'data', 'functions', 'fix', 'bug', 'mode'. The presence of 'fix', 'bug' among top keywords (based on TF-IDF score) implicitly suggests discussion of code defects. We experimented with all clusters and used the ones that improved the classifier performance.

Feature	**Source**	**Type**
Total number of messages in patch discussion	LKML messages	social
Probability of patch reverting message in patch discussion	LKML messages	social
Increase of patch files in the last eight months	LKML messages	social
Average BERT score of messages in patch discussion	LKML messages	social
Number of messages in patch discussion in cluster 12 (embedding-based clustering)	LKML messages	social
Number of messages in patch discussion in cluster 12 (embedding-based clustering)	LKML messages	social
Number of messages in patch discussion in cluster 0 (keywords-based clustering)	LKML messages	social
Average number of lines of code inserted across patch files	Patch diff	code
Average number of lines of code deleted across patch files	Patch diff	code
Average number of files changed	Patch diff	code

Table 1. Features, their source, type and code names

For training we used LKML messages between January 1 December 31, 2020. The training set contained 317K messages. We identified 30K patch discussions (mail thread about one patch), 2749 of which were later reverted. Information about patch reversal from 2020 was collected through the first half of 2021.

We compared ML model performance using receiver operating characteristic (ROC) curve and area under the ROC curve metric (AUC). Summary of the performance is in Fig. 3. The random forest classifier had the highest AUC of 0.78, allowing more than 60% of reverted patches to be identified while falsely predicting less than 10%. We used random forest classifier predictions to compare the number of patches reverted in the test period with the simulation.

6 Experiment setup and results

We train our simulation on 212,000 LKML messages between January 1 and June 30 of 2020 from approximately 5,000 authors. The test period for both simulation models was from July 1 2020 to June 30 2021. The two simulation models each generated 340,000 LKML messages. We used the first half of the year 2021 to estimate cognitive workload on reviewers. The first half of the test period in the simulation was used to generate patches (LKML patch discussions), and the second half of the test period was used to measure the number of patches reverted across maintainer groups (Fig 4). Simulation models generated LKML messages with new patch proposals. Then these patches were reviewed, with some being accepted and some later reverted.

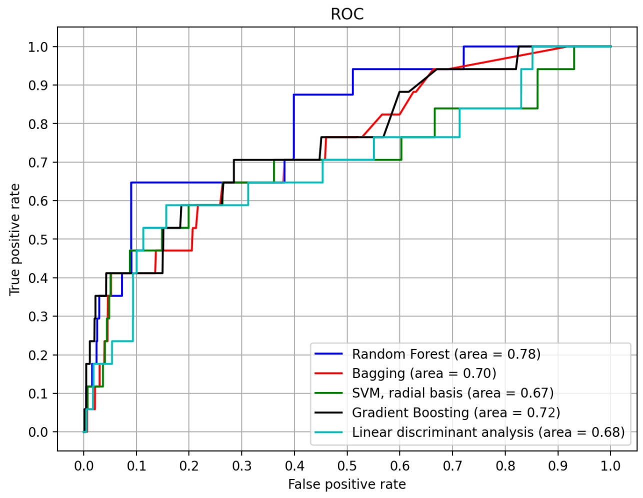

Fig. 3. ROC and AUC for ML models predicting patch reversal.

Unlike the simulation models, the ML models only predict if a patch is going to be reverted based on the features of the patch. These features are computed on activity six months earlier in the maintainer group in which the patch is proposed. We used LKML data from the test period (the first half of 2021) and the random forest classifier to predict patch reversal.

In order to compare how the number of proposed patches affects the workload of reviewers and maintainers we used two measurements: 1) the number of patches proposed per maintainer group, and 2) the number of patches reverted during the same period. Plots in Fig. 4 a) shows that the number of patches proposed per maintainer group is associated with the number of patches reverted during the same period. Plots in Fig 4 b) indicate that the percent of patches proposed per maintainer group is associated with the number of patches reverted during the same period.

Both simulation models and the random forest ML classifier (ML prediction in Fig. 4) predict a similar number and percentage of reverted patches. The number of reverted patches increases with the number of patches proposed in each maintainer group. The plots only show the top 20 (by number of proposed patches) maintainer groups. All models show a positive correlation ($0.60 \leq r^2 \leq 0.74$) between the number of proposed patches and the number of reverted patches. This means that busier maintainer groups with a high volume of new patches have higher volumes of reverted patches. There is also a weak positive correlation ($0.24 \leq r^2 \leq 0.57$) between the number of patches proposed in each maintainer group and the percent of patches reverted. All models capture similar levels of correlation relative to the ground truth.

We used the Kolmogorov-Smirnov Goodness of Fit Test (K-S test) to compare how close our models' predictions are to the ground truth. The KS-test statistic of the number of patches reverted per maintainer (Table 2) shows that the simulation with patch review process model is closer to the ground truth distribution than both the ML model, and the base simulation without process model. The D statistic for the process model is 0.0.01618, where smaller values indicate that samples are likely from the same distribution. In Fig. 5 we also compared distributions of the number of messages per reverted patch. Both simulation results show similar results but differ from the ground truth.

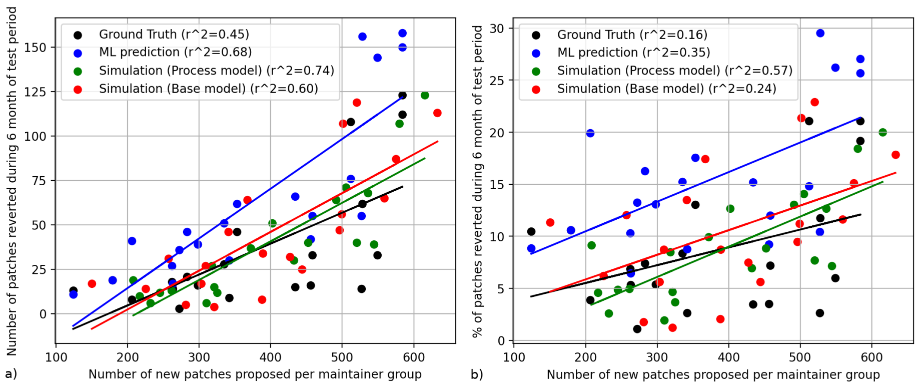

Fig. 4. a) Number of patches reverted during 6 month of test period, b) Percent of patches reverted during 6 month of test period

	D statistic	p-value
ML model	0.02107	0.7554
Simulation (Process model)	0.01618	0.9524
Simulation (Base model)	0.02157	0.7298

Table 2. KS-test statistic of the number of patches reverted per maintainer group predicted by ML and simulation models.

7 Discussion

We described an agent-based model of collaborative development of open-source software that uses an explicit model of the process by which software patches are proposed, reviewed and accepted or rejected. The model was compared to an ablated version without the process model, based on how well it predicts the number of proposed patches and the number of reverted patches, and both were compared to ML models that do not use simulation. Both simulation models were

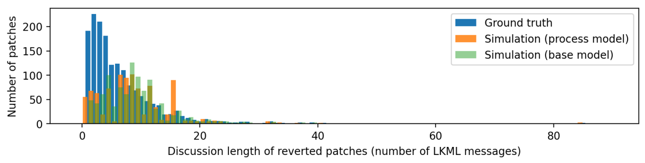

Fig. 5. Distribution of the number of messages per reverted patch.

designed as general purpose agent-based models reproducing user interactions via LKML messages and used a shared set of assumptions about the impact of workload on the quality of patches that matched our ground truth data. Both simulation models showed a closer fit to ground truth data than our best ML prediction, using a random forest, and the process-aware model showed a slightly closer fit than the ablated model.

While there are indications that the relatively simple models of the effect of workload on quality and of the patch discussion process have predictive value, we plan to improve them based on other observations of the community. For example, the disparity in the number of short patch discussions predicted as shown in Figure 5 may be addressed by introducing more variables related to the transition probabilities used in the process model, although the relationship between this length and patch quality is unclear.

Many other predictive tasks can also be used to test the efficacy of the agent-based simulation and of its process model that are beyond the scope of this paper, including predictions of the evolution of developers into reviewers and maintainers, or of their rising or falling influence within the community, and predictions about the changing levels of activity in different areas of the code base. While the Linux kernel and its public discussion list provides a rich source of data, we plan to supplement this in future work with subsidiary repositories and mailing lists, and other public discussion forums.

Simulation models have a few limitations compared to ML models. For instance, it is hard to use simulation models to predict whether an individual patch will be reverted. ML models only need a feature vector to answer this question. On the other hand, simulation models can answer more sophisticated analytical questions about community performance under different conditions. For example, how well maintainers, reviewers and developers can handle a sudden influx of patch proposals.

To compare the predictive powers of simulation models with non-simulation models we developed several ML models using standard algorithms. Although it is likely that with more data and better feature sets ML models would show improved performance, in these experiments our ABM led to a closer match to ground truth data.

The model of cognitive workload that we implemented to estimate the probability of patch reversal in the process simulation model also has some limitations. LKML messages only show whether a patch was reverted; however, it is hard to estimate to what extent this was due to the review process (experience and the number of reviewers) or due to the experience of developers contributing the patch. Alternative parametric models estimate the impact of these two factors on the probability of patch reversal in the environment where developers, reviewers and maintainers have deal with extensive multitasking (e.g. [11, 13]). These models often rely on more detailed data about work activity. For example, work logs, hours committed to produce a patch or some unit of code, etc. More advanced feature sets can also improve patch reversal predictions. We plan to explore models of this kind in future work.

References

1. Linux kernel git repository, https://git.kernel.org/
2. Linux kernel mailing list archive, https://lkml.org/
3. Dash agent-based modeling framework, https://github.com/isi-usc-edu/dash/
4. List of linux kernel maintainers and how to submit kernel changes, https://www.kernel.org/doc/linux/MAINTAINERS
5. Blythe, J., Bollenbacher, J., Huang, D., Hui, P.M., Krohn, R., Pacheco, D., Muric, G., Sapienza, A., Tregubov, A., Ahn, Y.Y., Flammini, A., Lerman, K., Menczer, F., Weninger, T., Ferrara, E.: Massive multi-agent data-driven simulations of the github ecosystem. In: International Conference on Practical Applications of Agents and Multi-Agent Systems. pp. 3–15. Springer (2019)
6. Blythe, J., Tregubov, A.: Farm: Architecture for distributed agent-based social simulations. In: Massively Multi-Agent Systems II: International Workshop, MMAS 2018, Revised Selected Papers. pp. 96–107. Springer (2019)
7. van Dam, K.H., Nikolic, I., Kukszo, Z.: Agent-Based Modeling of Socio-Technical Systems. Springer (2013)
8. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding (2018). https://doi.org/10.48550/ARXIV.1810.04805, https://arxiv.org/abs/1810.04805
9. Jiang, Y., Adams, B., German, D.M.: Will my patch make it? and how fast? case study on the linux kernel. In: Working Conference on Mining Software Repositories (MSR) (2013)
10. Kovanis, M., Porcher, R., Ravaud, P., Trinquart, L.: Complex systems approach to scientific publication and peer-review system: development of an agent-based model calibrated with empirical journal data. Scientometrics **106**(2), 695–715 (2016)
11. Meyer, A.N., Fritz, T., Murphy, G.C., Zimmermann, T.: Software developers' perceptions of productivity. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 19–29 (2014)
12. Spasic, B., Onggo, B.S.S.: Agent-based simulation of the software development process: a case study at avl. In: Winter Simulation Conference (2012)
13. Tregubov, A., Boehm, B., Rodchenko, N., Lane, J.A.: Impact of task switching and work interruptions on software development processes. In: Proceedings of the 2017 International Conference on Software and System Process. pp. 134–138 (2017)
14. Weinberg, G.M.: Quality software management (Vol. 1) systems thinking. Dorset House Publishing Co., Inc. (1992)