Developer Oriented and Quality Assurance Based Simulation of Software Processes

Verena Honsel, Daniel Honsel, Jens Grabowski, and Stephan Waack
Institute of Computer Science
Goldschmidtstr. 7
University of Goettingen
{vhonsel,dhonsel,grabowski,waack}@cs.uni-goettingen.de

Abstract

Software process planning involves the consideration of process based factors, e.g., development strategies, but also social factors, e.g., collaboration of developers. To facilitate project managers in decision making during the project, we develop an agent-based simulation tool which allows them to test different alternative future scenarios. For this, it is indispensable to understand software evolution and its influences. We cover different aspects of software evolution with models tailored towards specific questions. For the investigation of system growth, developer networks and file dependency graphs, we performed two case studies of open source projects. This way, we infer parameters close to reality and are able to compare empirical with simulated results.

1 Introduction

In software process planning decision making is a hard but important task for project managers. It can be of great help to have tool support, with which the manager can test the interplay of project parameters and resulting evolutionary scenarios. Relevant parameters may be the number of developers, their bugfixing effort, and the expected development time when deciding, e.g., about the team constellation. The bugfixing effort depends on their roles, e.g., maintainers fix more bugs. This process is iteratively repeated until the project manager gets sufficient information. The intended feedback loop is depicted in Figure 1.

To build an agent-based simulation tool aiding software managers in the planning of software development, it is important to get a deep understanding of software evolution processes. Several factors influence *how* the software evolve, *what* evolves, and *why* it evolves. According to Lehman [2], finding answers to these questions are the research directions in software evolution. To reduce complexity and parameters, we build different models reflecting different shades of software evolution and related development processes. Since humans – in the shape of developers, users, and testers – constitute a big driver of software evolution, it is reasonable to approach the simulation of software processes agent-based. Agents are autonomous individuals with a behavior specified by certain rules [3]. Developers can be considered as active agents changing the passive agents, i.e., software entities.

When tracing aspects of software evolution, we first have to identify influencing factors concerning the question under investigation. Thus, we learn from the past in form of analyzing open source software repositories, which by itself became a large research topic in recent years (e.g., MSR [1]). In our approach we combine software quality assurance issues with social and process controlled factors influencing software development. For this, we are interested in examining the contribution behavior of developers as well as the nature of changes and related error-proneness. The knowledge we gain from mining is then transfered into our agent-based simulation model so that we retrieve a concrete instance constructed for answering the spe-

In: A.H. Bagge, T. Mens (eds.): Postproceedings of SATToSE 2015 Seminar on Advanced Techniques and Tools for Software Evolution, University of Mons, Belgium, 6-8 July 2015, published at `http://ceur-ws.org`

[1]http://2016.msrconf.org/ [last visited: 10.03.2016]

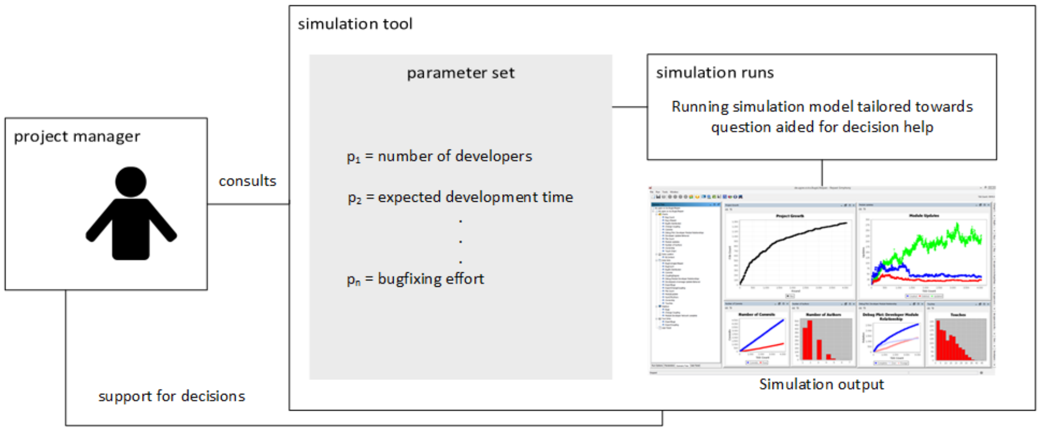

simulation tool
project manager
consults
parameter set
p₁ = number of developers
p₂ = expected development time
pₙ = bugfixing effort
simulation runs
Running simulation model tailored towards question aided for decision help
Simulation output
support for decisions

Figure 1: Feedback loop for project managers [1].

cific evolutionary question under investigation. In this paper, we summarize our recent research, which considers system growth, developer collaboration and behavior, and the evolution of software changes. This paper presents a publication summary of our papers [4], [1], and [5].

2 Related Work

Only few approaches exist employing simulation in the context of software evolution and the prediction of it. The work most related to our approach is the work of Smith et al. [6] which proposes to use Agent-based simulation in the context of software evolution. There, as well the developers serve as active agents and the software entities (here modules) constitute the passive agents. They also have requirements as a primary stage of software modules. As metrics they consider complexity and fitness (of purpose) of modules, which are changed by the actions of developers. In contrast to their work, the environment is not defined on a grid, where the developers can fall down when moving along. Instead we use networks, which have the additional advantage to store dependencies between entities. The quality, in the work of Smith et al. modeled by the fitness, is modeled by the bug distribution in our case, but we do not consider complexity at the moment.

Other studies touching the topic are for example the work of Wagstrom et al. [7] and Andersson et al [8].

3 Approach

In this section, we describe the background and methods which represent the foundations of our work. We comment on methods used for software mining and explain the underlying agent-based model of our approach.

3.1 Software Mining and Analysis

For the estimation of the simulation parameters, we examine open source software projects, which are a gold mine for researchers interested in software evolution and software mining. Since we are interested in different facets of the software development process, we collect data from projects, for which the information of commit logs, issue tracking systems, and mailing lists are available. Once the project is selected, we use tools from data mining and analysis, machine learning, and visualization to first understand it and later observe behavioral rules from it. For analyses we use the tools R [9] and Weka [10]. The observed rules and information then serve us as input for our simulation model.

3.2 Agent-Based Simulation Model

The current agent-based simulation model for software evolution is an extension of our previous publication [4] which considers only the system growth of the software under simulation. For modeling and simulation purposes we use Repast Simphony [2].

The model depicted in Figure 2 contains the environment which knows all other instances and is responsible for the creation of a configured number of developers at simulation start-up. Furthermore, the environment instantiates bugs at scheduled points in time and assigns them to randomly selected software entities, e.g., files, classes, or modules. The developer is responsible for creating, updating, and deleting entities. For the estimation of parameters we used K3b [3] in our initial case study. Through the mining process we have recognized four different types of developers.

The *Core Developer* is the initial contributor being familiar with many entities and performing most

[2]http://repast.sourceforge.net/ [last visited: 10.03.2016]
[3]http://www.k3b.org/ [last visited: 10.03.2016]

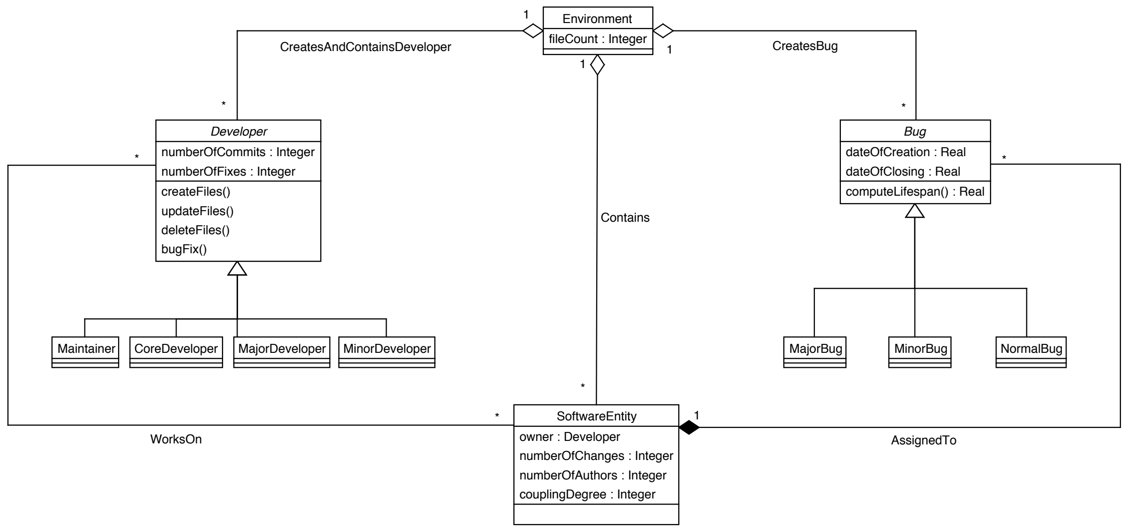


Figure 2: Simulation model [1].

commits. The *Maintainer* is a person who does primarily maintenance work, i.e., he fixes a large number of bugs. Therefore, we assume he has good knowledge about the entire project. The *Major Developer* knows specific areas of the project and fixes most of the bugs occurred in entities known by him. The *Minor Developer* executes less than 100 commits and performs less bugfixes. They might be specialists who only implement one specific task or feature. In K3b there is one core developer, one maintainer, 17 major developers, and 106 minor developers.

To model dependencies between the agents, we have created three networks. One to represent dependencies between developers and software entities, one stores information about bugs and the modules they are assigned to, and one represents dependencies between software entities that are changed together several times. Following, the networks are briefly summarized:

- *DeveloperEntityNetwork*: This network represents the dependencies between entities and developers. An edge is added if a developer creates an entity or if a developer changes an entity, that has not been created by him. Hence, this network also provides the number of authors.

- *BugEntityNetwork*: After the environment created a new bug an edge is added to this network. The edge contains information whether a bug is fixed or not. In future models an edge may contain additional information about the bugs, e.g., the number of fixing attempts or if a bug is reopened.

- *ChangeCouplingNetwork*: This network represents dependencies between software entities that are changed together, including the number of changes.

The creation and deletion of entities is responsible for the growth of the software under simulation. The growth depends on the number of developers, the desired system size, and the simulation time. Since the lifespan of K3b is 4044 days, we have 4044 simulation rounds. We assume that the number of entity changes follows a geometric distribution [4].

Table 1: Developers' average commit behavior in K3b.

Developer	#Commits	#Fixes
Core	3384	869
Maintainer	499	152
Major	445	79
Minor	10	4

Table 2: Developers' average add, update, and delete behavior in K3b per commit.

Developer	Add	Update	Delete
Core	0.6	5.5	0.4
Maintainer	0.9	3.5	0.3
Major	0.2	5.2	0.4
Minor	0.1	2.0	0.04

The likelihood of creating and deleting entities decreases with the increasing system growth. The probabilities are restricted through the project size and adjusted for each developer type with respect to the average commit behavior shown in Table 1 and the

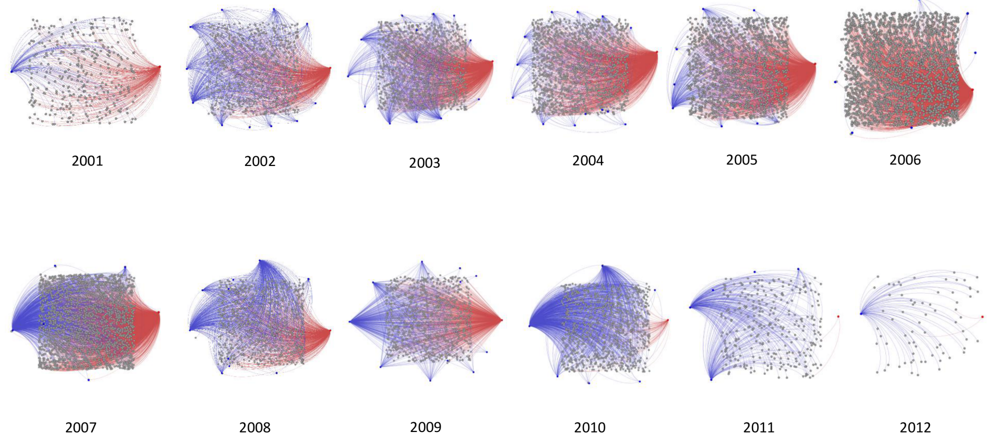


Figure 3: Yearly developer-file networks of K3b [4].

average add, update, and delete behavior per commit presented in Table 2. We assume that it is more likely that developers update entities they already know.

4 Performed Analysis and Case Studies

We briefly summarize two of our case studies and results in this section. These studies include the system growth in number of files, developer collaboration depicted in developer-file networks, and the evolution of software dependencies represented in file networks based on change coupling. An approach for the learning of developer experience and project involvement is also presented.

4.1 System Growth and Developer Collaboration

One aspect of our preliminary case study [4] is the growth of software systems. We measure growth in number of files, which is reflected by the creations, modifications, and deletions the developers perform. For this purpose, we selected K3b with a development time of over ten years, 125 developers and more than 6000 commits. We observed a super-linear growth trend for K3b and used this to build a statistical model for the growth based on changes made by developers. Using geometric distributions for file creations, modifications, and deletions, we were able to reproduce the system growth in number of files of K3b validated by comparing empirical and simulated results. The simulated curve fits the trend as well as the concrete values given the parameter set for K3b.

Moreover, we build developer-file networks, where a dependency between a developer node and a file node is added, if the developer worked on that file. The evolution of the graph depicted in Figure 3 shows that there is one main contributor who is the project creator (red node), whose central status is inherited by its maintainer (blue node on the left) after 2006. Moreover, we have a low modularity factor in 2006 and 2012, i.e., the network cannot be modularized into clusters. There the work depends too much on a certain developer. How and why these networks evolve for other projects is of interest for simulating developer collaboration behavior. As Foucalt et al. [11] stated, developer turnover can have a high impact on software quality. To identify such turnover patterns, could also improve the simulation of software processes.

4.2 Software Dependency Analysis

In our latest work [1], we analyzed change coupling dependency graphs to understand the evolution of file dependencies. The change coupling [12] degree describes how often software entities are changed together. By calculating the average degree as well as the average weighted degree over the time, we trace the evolution of the files not only in terms of the amount of dependencies to other files, but also in terms of the intensity of their relationship. For this, we used K3b for the estimation of parameters and Log4j [4] for the validation of our results. The two projects are alike in the number of files and duration, but differ in the effort spent by the developers. e.g., K3b comprises more developers, especially minor developers.

For the estimation of parameters we build change coupling dependency graphs from commit logs. If a file is changed together with another file more than twice, an edge is created in the network. To describe

[4]https://logging.apache.org/ [last visited: 10.03.2016]

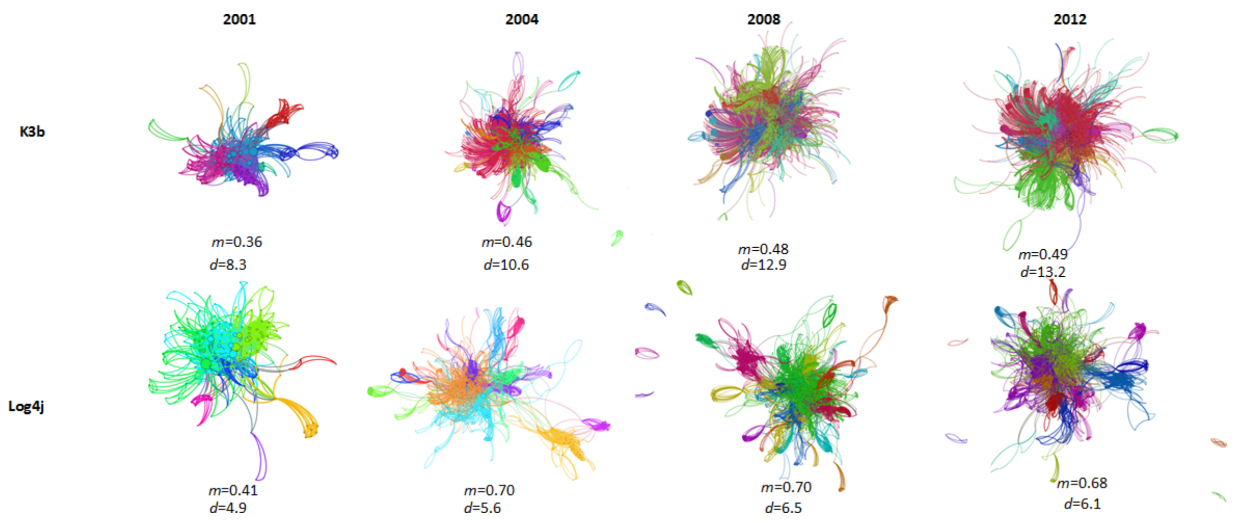


Figure 4: File dependency graphs of K3b and Log4j including network modularity m and average degree d [1].

the evolution of software dependencies in the simulation we compared the resulting graphs for each year in terms of the degree, modularity, and diameter. The degree of a node (file) exposes the importance of the node. The modularity indicates how good a network can be divided into clusters, whereas the diameter defines the maximum shortest path between each pair of nodes [13].

In Figure 4 the evolution of K3b's as well as Log4j's file dependency network for selected years is illustrated. In the case of K3b, we have a quite high average degree, which means that there are less weakly connected entities and the networks remain quite dense. The modularity instead is lower than in the case of Log4j, so that the separation into clusters is worse. For Log4j several small independent clusters are visible, which constitute for example tests.

The empirical behavior (red) of the average change coupling degree is shown in Figure 5. To model this trend we used linear regression and retrieved the best fit for a second order model (black) with an adjusted R-squared value of 0.97 [1]. Also the simulated average degree of the software agents is depicted. By comparing the real with the simulated behavior, it is recognizable that the simulation exhibits a similar trend but is about three too low.

For validation we tried out the simulation built on the knowledge gained from K3b with a changed parameter set according to properties of Log4j. There we have one core developer, five major developers, and fourteen minor developers. With the adapted distribution of developer types and size of the system, we were able to simulate growth trends and network properties.

In Figure 6 the empirical, fitted, and simulated average coupling degree over the years is pictured. Therefore, the simulation works for projects which are similar in size and duration. The projects under examination are also written in the same programming language. We also validated the square trend and the

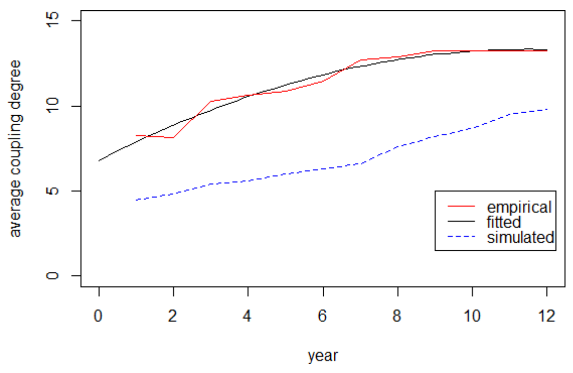


Figure 5: Average empirical and simulated coupling degree of K3b [1].

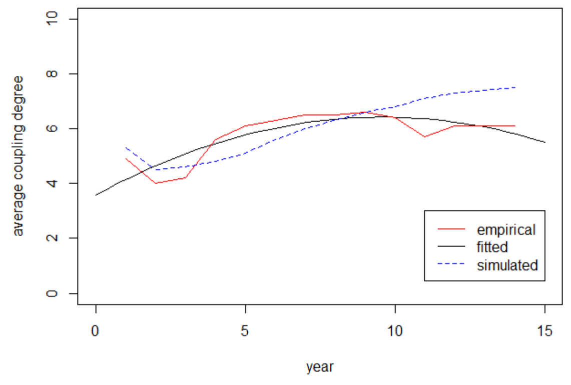


Figure 6: Average empirical and simulated coupling degree of Log4j [1].

average coupling degree for Log4j.

Since the roles of developer types are currently static and we observed in our work, e.g., in the evolution of developer-file networks (Figure 3) that the roles and importance of developers can change over time, we study the developer contribution behavior in more detail.

4.3 Developer Contribution Behavior

Since our work exposed the need for a more fine-grained model of developer behavior, we created a learning model, which helps us to understand developer contribution behavior and related experience. A first improvement of our simulation was to introduce developer types, which are manually classified according to their commit and bugfixing behavior. The contribution behavior of developers complies with their personal status of experience and involvement in the project. In the analysis of contribution behavior only the output of this status is visible. To retrieve information about the underlying states, we employ Hidden Markov Models (HMMs), which are stochastic models used for discrete time observations. In doing so, we hope to gain valuable insights for the refinement of developer types.

The general method as described in [5] takes three sources of data into account: version control data in form of developer commits and bugfixes, bug tracking system information in terms of bug comments, and mailing list data by the number of threads opened and answers for each developer. For every contributor in the project these learning activities are collected for each month. This requires a HMM which can handle multi-dimensional observations since we have four observations for each point in time. To map this into a comprehensible format, we classify these observations by using a threshold learner into low, medium and high. When filtering developers with less than twenty commits, ten active contributors remained for the project Rekonq[5].

In Figures 7a - 7d [5] the monthly contribution behavior as well as the monthly mailing list activities of these developers are visible. The project points out one main contributor (dev 1). For the communication activity displayed in Figure 7c and 7d it is shown, that also other developers play an active part there. As an explanation one can think of less experienced developers asking for help. The retrieved thresholds necessary for the HMM learning input are listed in Table 3 and Table 4 [5].

A developer needs for example at least 14 commits and no bugfixes to contribute with a medium activity and at least 35 commits and 6 bugfixes to contribute in a high manner. When the training of the HMMs for each developer is finished, we get the corresponding transition matrix containing the probabilities to switch between the different learning stages. Via the Viterbi algorithm [14] we are able to get the most probable sequence of learning states which produced the observation sequence.

[5]https://rekonq.kde.org/ [last visited: 10.03.2016]

Table 3: Thresholds for classifying contribution and communication behavior of developers.

	Contributions	Communication
Threshold	Commits/ Bugfixes	Opened/ Responses
low	$< 13.8/0$	$< 0/11$
medium	$\geq 13.8/0$ $< 34.5/5.5$	$\geq 0/11$ $< 0/27.5$
high	$\geq 34.5/5.5$	$\geq 0/27.5$

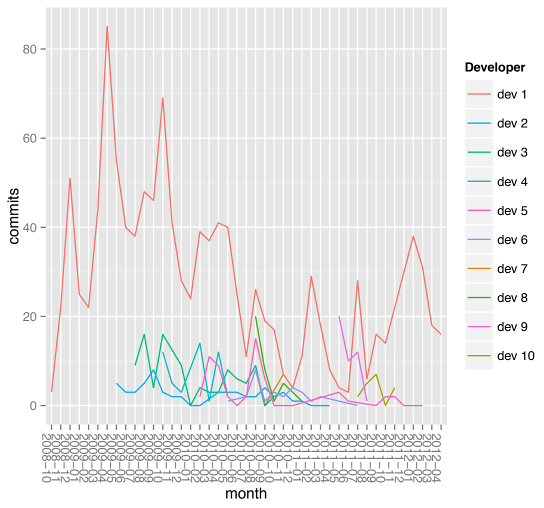
Table 4: Thresholds for classifying bug activity of developers.

	Bug Activity
Threshold	Bug Comments/Bug Reports
low	$< 20.7/0$
medium	$\geq 20.7/0$ $< 62.1/0$
high	$\geq 62.1/0$

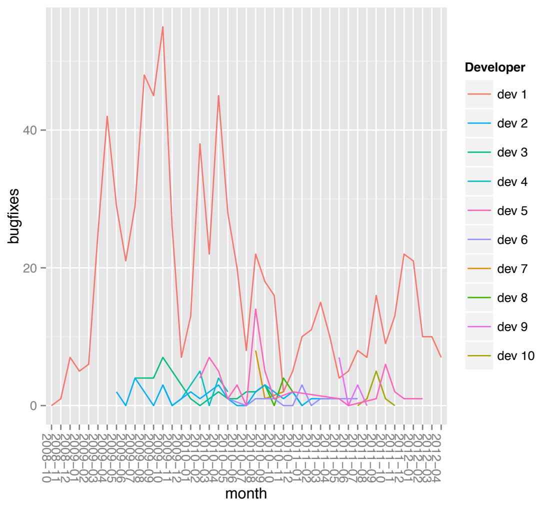
5 Conclusion

In our case studies we showed the feasibility of agent-based simulation for software processes. We experienced that some facts as the growth and the general commit behavior can be modeled well. But when it comes to phenomena of software evolution, like the developer turnover mentioned in Section 4.1, which are project-specific, it is more complex to describe. Therefore, it could be of help to generate developer profiles storing more information about them, and which can serve the project manager as indicator for such an incidence. Moreover, the dependencies among agents are intricately describable. Although we were able to simulate the co-changes of K3b successfully, er lost precision in the results for Log4j. Probable solutions are described in the next section. Summarily, simulation turns out to be a good method for software evolution, but the whole process including selected parameters and dependencies needs to be considered carefully and describing it accurately is a strenuous process.

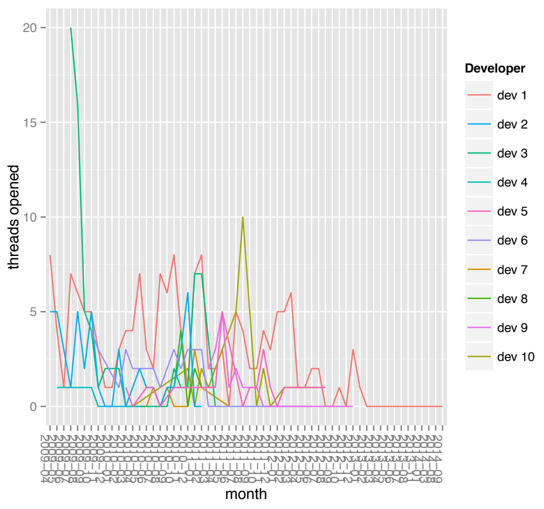
During our experiments in simulating software processes, we observed problems due to the lack of development phases, which results in a more linear rep-



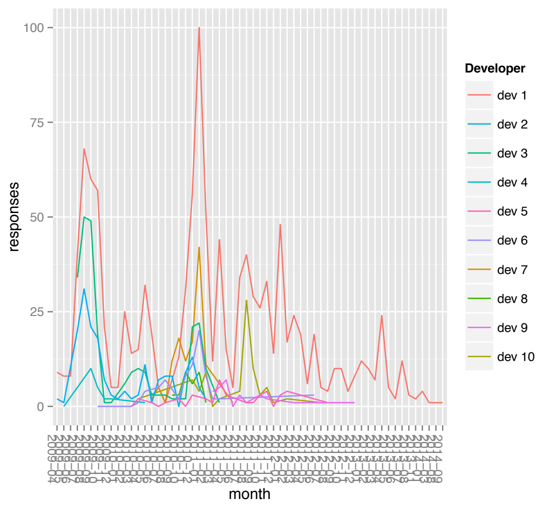
(a) Commits.



(b) Fixed bugs.



(c) Opened threads.



(d) Responses.

Figure 7: Monthly contribution behavior and mailing list activities of the active developers contributing to the project Rekonq.

resentation of, e.g., system growth, on the simulation side. The need of phases is also relevant for the behavior of developers. Although there are different types of developers in the simulation, they spend the same effort and do not learn from their experiences.

6 Future Work

To include different learning types of developers, we plan to implement the knowledge learned by the HMMs into the simulation. From this, we hope to refine the developer types and incorporate development strategies according to the developers' behavior. This model will be transfered into a learning model for development phases like initial, development, or maintenance. This originates from the fact that at different points in time certain actions are more likely, e.g., in the beginning of a project creations are more likely, whereas in the end or before a release bugfixes are common. How suitable parameters look like and if this presents a promising approach for the inclusion of development phases in the simulation, is an open question for us.

Furthermore, we plan to improve the strategy for the bug introduction. With every change in the software a bug can be introduced with a certain probability. This probability depends on factors like the experience of the author of the change, the number of previous changes, and the complexity of the artifact under change. Since we already simulate such factors, we plan to build a heuristic (e.g., a decision tree) using this information which defines the bug introduction probability.

Since we also have different graphs describing software evolution in our simulation, we plan to use them for software quality prediction. Bhattacharya et al. [15] showed the correlation between different kinds of networks concerning software evolution (displaying developer collaboration, module dependencies, and function calls) and quality factors like defects and maintenance effort. For such an investigation we need also networks displaying the structure of the software in more detail. For this, we are currently working on the mining of abstract syntax trees (ASTs) retrieved from the source code and examining the evolution of them. This way, we hope to get a fine-grained picture of software evolution resulting in a simulation which puts these networks together and simulate the effect on software quality.

For the evaluation of empirical and simulated networks we plan to use exponential random graph models (ERGMs) [16], which allows us to get a structural representation of graphs, which facilitates the comparability. Instead of a set of metrics, we get a whole describing picture of structural properties.

6.0.1 Acknowledgements

We would like to thank the simulation science center Clausthal-Göttingen (SWZ), that funded parts of our work.

References

[1] V. Honsel, D. Honsel, J. Grabowski, and S. Waack. Mining software dependency networks for agent-based simulation of software evolution. In *Proceedings of the 4th International Workshop on Software Mining (SoftMine)*, 2015.

[2] M.M. Lehman and J.F. Ramil. Towards a theory of software evolution - and its practical impact (working paper). In *Invited Talk, Proceedings Intl. Symposium on Principles of Softw. Evolution, ISPSE 2000, 1-2 Nov*, pages 2–11. Press, 2000.

[3] C. M. Macal and M. J. North. Introductory tutorial: Agent-based modeling and simulation. In *Simulation Conference (WSC), Proceedings of the 2011 Winter*, pages 1451–1464. IEEE, 2011.

[4] V. Honsel, D. Honsel, and J. Grabowski. Software process simulation based on mining software repositories. In *Proceedings of the Third International Workshop on Software Mining*, 2014.

[5] V. Honsel. Statistical learning and software mining for agent based simulation of software evolution. In *Doctoral Symposium at the 37th International Conference on Software Engineering*, 2015.

[6] N. Smith and J. F. Ramil. Agent-based simulation of open source evolution. In *Software Process Improvement and Practice*, pages 423–434, 2006.

[7] P.A. Wagstrom, J. Herbsleb, and K. Carley. A Social Network Approach To Free/Open Source Software Simulation. *Proceedings of the 1st International Conference on Open Source Systems, Genova, 11th-15th July*, 2005.

[8] C. Andersson, L. Karlsson, Jo. Nedstam, M. Hst, and B. I. Nilsson. Understanding software processes through system dynamics simulation: A case study. In *ECBS*, pages 41–. IEEE Computer Society, 2002.

[9] R. Ihaka and R. Gentleman. R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[11] M. Foucault, M. Palyart, X. Blanc, G. C. Murphy, and J.-R. Falleri. Impact of developer turnover on quality in open-source software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 829–841, New York, NY, USA, 2015. ACM.

[12] T. Ball, J. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk. In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, volume 11, 1997.

[13] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75 – 174, 2010.

[14] L. R. Rabiner. Readings in speech recognition. chapter A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, pages 267–296. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[15] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 419–429, Piscataway, NJ, USA, 2012. IEEE Press.

[16] David R. Hunter, Steven M. Goodreau, and Mark S. Handcock. Goodness of fit for social network models. *Journal of the American Statistical Association*, pages 248–258, 2008.