An Unified Representation of Source Code Authoring Workflows

Dmitrii Timofeev[1] and Alexander Samochadin[1,2]
[1]*Mobile Device Management Laboratory, Peter the Great St. Petersburg Polytechnic University, Polytechnicheskaya 29, St. Petersburg, Russia*
[2]*Higher School of Software Engineering, Peter the Great St. Petersburg Polytechnic University, Polytechnicheskaya 21, St. Petersburg, Russia*

Keywords: Process Modeling, Source Code Analysis, Software Development Process, Source Code Authoring.

Abstract: Existing approaches to modeling software development processes mostly deal with high-level processes at the level of project management. There are specific tasks that involve the analysis of processes at the level of writing and modifying the program code, but they lack a common reusable modeling framework. We suggest that a model of source code editing workflow would be beneficial for many tasks, from defect prediction to teaching programming to novices. We propose a unified approach that combines several levels of annotations, from keyboard events to task tracker issues and project planning.

1 INTRODUCTION

Software development comprises multiple workflows that result in a set of artifacts, with the main being the source code. Its principal use is the compilation, optimization, and execution of the program. At the same time, the source code serves an input for other tasks, including authorship detection and defect prediction. Reading source code, and writing it to be read by others, also plays a significant role in teaching programming.

The defect prediction task is usually defined as a supervised learning problem where the goal is to associate source code fragments with the probability or an error-proneness class of a defect. There are two kinds of features for the learning problem: code metrics like method size or complexity of a control flow graph, and process metrics, that measure the properties of the development process. Examples of process metrics are the frequency of changes and the number of developers working on the same piece of code. As Rahman and Devanbu (2013) have shown, process metrics are significantly more informative than code metrics for the defect prediction task. The standard practice to collect process metrics is to parse the commit history in the project version control system, and to associate commits with defect reports in the bug tracker. Unfortunately, the commit history provides only the data about the state of the code at the specific checkpoints and hides the way the programmer writes the code. At the same time, the code editing workflow itself often contains clues about possible bugs. For example, a well-known source of defects is the copy-paste programming that increases the probability of bug duplication and context-related errors (Kim et al., 2004). As Hou et al. (2009) note, static analysis algorithms cannot provide the same copy-paste clone detection quality as the dynamic process analysis methods. Integration of these low-level process metrics would improve the quality of defect prediction systems.

Another application of process analysis is teaching the novice programmers. Learning the right workflows of writing and maintaining the source code is a necessary part of programmers education. While checking the coding standards conformance and paying attention to compiler warnings helps to get rid of several bad habits the students tend to develop, the more subtle code editing patterns, like the excessive copy-pasting mentioned before, are much harder to detect without surveying how the student works on the task. The typical workflow employed by a programmer also classifies developers by experience level and preferred style. Several styles may be more or less desirable in a production environment than others. For example, developers may tend to invent new solutions, to adopt standard solutions from their knowledge base, or to search and reuse solutions from sites like StackOverflow[1].

Logging and analysis of actions performed by a

[1]https://stackoverflow.com/

Timofeev, D. and Samochadin, A.
An Unified Representation of Source Code Authoring Workflows.
DOI: 10.5220/0007229902280232
In *Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2018) - Volume 3: KMIS*, pages 228-232
ISBN: 978-989-758-330-8

developer in an integrated development environment (IDE) is a widely used technique. At the same time, each tool implements this method independently and tune it to a specific task. In this position paper, we discuss an approach to a code editing process model and a set of action capturing tools that could be reused for solving particular problems like defect prediction, developer performance analysis, or teaching good programming practices. Our approach is to represent different aspects of the software development process as a hierarchical set of annotated events in a single time scale.

2 RELATED WORK

The document authoring workflows have been studied in multiple research areas, from psychology and human-computer interaction (HCI)to software engineering. A particular process that is widely represented in psychology and HCI papers is writing or modifying documents using text editing software. In these cases, researchers are most interested in the ways users interact with a computer system, their mental models, and the factors that influence the interaction, like user interface complexity (Card et al., 1980), user experience (Rosson, 1983), or interruptions (Burmistrov and Leonova, 2003). The experimental data is usually collected using video recording or by logging keys and user commands. The process is described with one or more quantitative characteristics, e.g., the number of operations per a time unit.

The most common workflow model used in these studies is the chronologically ordered sequence of actions. Burmistrov and Leonova (2003) replaced atomic operations with the events start and stop marks so that their model could describe nested and postponed activities. Polson and Kieras (1985) proposed a generative model where a set of production rules describes the user behavior. When the editing context matches a rule, this rule fires and the model generates a sequence of higher-level editing commands.

In software engineering, human behavior models are widely used for task automation and functional testing. Tools like Expect[2] and Selenium[3] allow developers to describe the interaction process as a program in a domain-specific or a general purpose programming language. When these programs are run, they interact with the system by analyzing its output and providing necessary input.

At the same time, the workflows that software developers themselves are using have been studied to a much lesser extent. Although there is a lot of research on project management, individual processes of writing the code aren't in focus. Well-known books on best programming practices (Hunt and Thomas, 1999; McConnell, 2004; Martin, 2008) deal mostly with software architecture and coding techniques. The process of programming is to some extent discussed by Carter and Sangler (1997). The book concerns with the mental models and strategies ("mapping" and "packing") that programmers use while working on the code.

The main problem of experimental studies of program authoring processes is their cognitive nature that makes difficult to trace workflow states and transitions as they are hidden from the observer. One way to overcome this difficulty is to ask programmers to explicitly describe the actions they perform, including purely mental tasks (Jeffries et al., 1981; Bennedsen and Caspersen, 2005). The possible states and transitions may be modeled as Markov processes (Kamma, 2014).

At the higher level, the workflow states may be linked to the tasks the developer solves. These tasks and their connections to the project issues (bugs or features) are usually tracked using a version control system. Each task results in a commit annotated with a description of the changes and with optional references to the issue tracker. Additionally, the source control repository provides data on the timeline and history of modifications and on developers involved. Commit logs seems to be the main source of information on programming workflows at the time present (Hassan and Holt, 2003; Hassan, 2009; D'Ambros et al., 2010; Rahman and Devanbu, 2013; Rubin et al., 2014).

The main artifact of the software developer's work is the source code. Modern IDEs provide rich code processing features including incremental parsing, type checking, and control flow graph analysis. Although these features are often available for plugins, there is just a limited number of process logging tools that make use of the source code, mostly for defect prediction. Although the set of employed features is usually limited, like the code fragments that have been copied and pasted (Kim et al., 2004; Hou et al., 2009), other structured features (adding and removal of code, modification of types or function arguments) may be worth logging as well (Lehnert, 2011).

As this review shows, the task of modeling workflows used by software developers, and other specialists whose operations are hard to observe due to their mental nature, is far from being solved. Although analysis tools exist for capturing specific process features (e.g., tracking code copying or extracting in-

[2]https://core.tcl.tk/expect/index

[3]https://www.seleniumhq.org/

formation from commit logs), events captured at different levels (key presses, program structure changes, code base modification, project status updates) are not linked together or used in combination.

3 MODELING THE CODE AUTHORING PROCESS

We propose to address the problem of modeling programming workflows by introducing a unified representation of programming activities.

There are three main aspects of the code authoring process.

- Activity: how the programmer writes the code.
- Contents: how is the code modified.
- Context: what is the purpose of these changes for the software project.

The process is recorded by tracking user actions. Different tasks require event logs to be recorded with a specific resolution. For example, in a defect prediction task, it might be interesting to search for methods or functions that were modified but not tested after the change. The event log should contain data about actions of types "modify the code," and "run unit tests". At the same time, to evaluate IDE ergonomics, we may need to count the number of mouse clicks required to complete a specific task.

To enable simultaneous representation of the events at several levels, we adopt the approach widely used in text and multimedia processing. We record the stream of directly observed actions and label event spans (continuous finite sequences of events) with annotations corresponding to higher-level events. For example, the action "run the application in debug mode" might match a span containing the specific sequence of mouse clicks and key presses. An annotation may be attached not only to low-level events but any other event span.

At the lowest level, we record atomic actions like key presses, mouse clicks, menu item activation, or window switching. We suggest using at least the following set of annotations (note that specific events here are examples, and the set of events at each level is not complete).

- Atomic operations: selecting an item in the current window, copying an item into the clipboard, pasting an item, running a program, starting a test suite and so on.
- Operations on text: selection, copying, pasting, inserting, removing and replacing the fragments of text. The logging of text operations is like the "track changes" feature of modern document editors like Microsoft Word.
- Operations on programs: modification of the program (module) syntax tree. Depending on the programming language, the set of operations may contain adding, removing, renaming of packages, classes, methods, functions and types, modification of their bodies and argument lists, adding or removing control structures (loops, conditions, sequences of operators).
- Operations on programs in an intermediate language: the same operations as in the previous list, but mapped not to the specific programming language but to an intermediate language to enable the creation of language-independent analysis tools. There may be several intermediate languages for different families of programming languages (procedural, functional, object-oriented, etc.)
- Process state: design, coding, debugging, testing.
- Project activity: implementing new features, fixing bugs, merging revisions, doing code reviews, committing to the source control repository.
- Project management activity: opening, modifying and closing tasks in a tracker, meeting, planning.

An example of this hierarchy is represented in the figure 1.

Note that annotations like "planning" or "meeting" may be added to the log even they don't have any corresponding low-level event (e.g., a developer is participating in a meeting and not using the computer until the meeting is over). To allow this behavior, we allow spans to be empty. An empty span is defined by its start and stop time labels.

There is another possible view of the same model. We may regard annotations as protensive events defined by specific content and a time span on the common time scale. Annotations may have explicit links to the inner events and vice versa, but they are not required. Events not linked to lower-level events correspond to annotations defined on empty spans.

To make the process representation less language-dependent, we suggest using a family of metalanguages. For most object-oriented programming languages we suggest representing the program structure in terms of FAMIX 3.0 metamodel (Ducasse et al., 2011). In this case, the program or component model is a tree whose nodes represent structural elements like packages, classes, interfaces, and methods. As the FAMIX metamodel does not include procedural facilities like operator and control structures, we may need to extend the common representation with the necessary features. In a similar way, higher-level

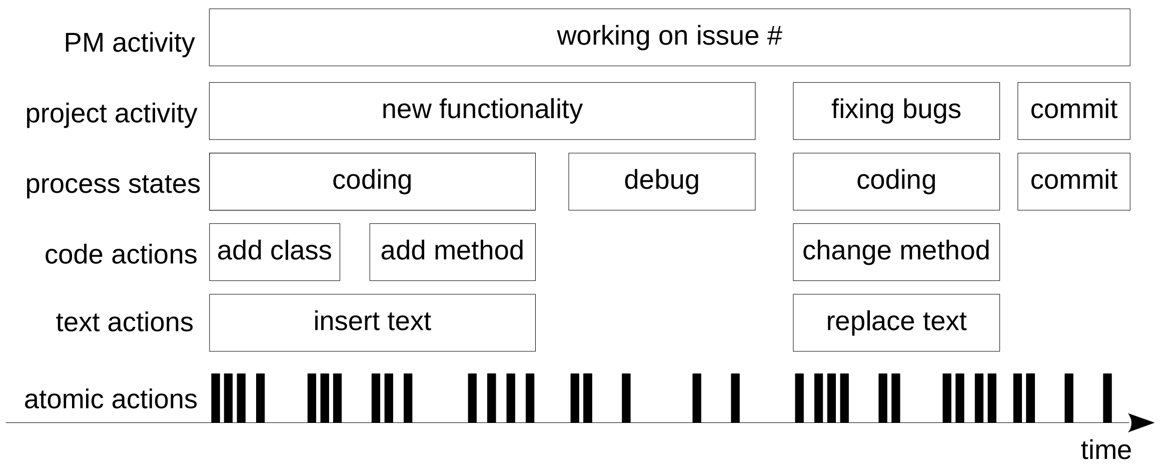

Figure 1: Combined log example.

events, e.g., project phases, may be mapped to OMG SPEM 2.0 metamodel (OMG, 2008).

For now, an early prototype has been developed that can log events generated by a user working in Microsoft Visual Studio IDE as well as lower-level events like key presses, mouse movements and clicks, and window switching. All these events are stored in a database using a common time scale. The current representation does not use explicit links to the inner events.

The problem of restoring these links is interesting by itself. Usually we have several sources of information about the event, for example, we may log the “open file” action either from a specific event published by Visual Studio or by parsing the sequence of pressed and released keys or mouse buttons. The main difference between these two approaches, besides the complexity of parsing the sequence of low-level keyboard and mouse events, is that parsing key presses restores links between higher-level and lower-level events while the independent event logging misses that information.

If the direct information about event linking, and about an event at a whole (e.g., on the process state level), is missing, there are three ways to infer it.

- Parse lower-level events to obtain links.
- Ask the user to annotate events.
- Restore links using machine learning techniques. In this case, the task may be stated as an event classification problem.

The latter case worth researching by itself, as the classification algorithm may be used to solve end-user tasks such as tracking user time, detecting the process state, or evaluating the user attention level and efficiency.

4 CONCLUSION AND FUTURE WORK

In this paper we propose an approach to modeling software development processes by tracking and annotating activities at several levels, from operations performed by an individual developer to the states of the entire projects. These events are linked together and also mapped to artifacts such as the software source code and commit messages. The model is designed to be used as input for higher-level analysis algorithms. We expect that the hierarchical log of developers’ actions and their timings will be beneficial for understanding software development processes, giving feedback to programmers and managers, and getting better results in program analysis tasks like defect prediction.

The concept described in this position paper is on the early stage of development. The primary research and implementation directions are the following.

1. Define an external representation of the process log.
2. Advance the current prototype by logging more high-level annotations, especially the annotations on program code level.
3. Define the set of statistical metrics to compare models, such as patch source frequency distribution or change localization.
4. Design and implement the model clustering algorithm to detect similar coding workflows and compare results to the externally estimated programmers’ efficiency.

ACKNOWLEDGEMENTS

This research is a part of the joint project by Intelin LLC (Moscow, Russia) and Peter the Great St. Petersburg Polytechnic University (St. Petersburg, Russia). This work is financially supported by the Ministry of Education and Science of the Russian Federation (state contract 03.G25.31.0247 from 28.04.2017).

REFERENCES

Bennedsen, J. and Caspersen, M. E. (2005). Revealing the programming process. In *ACM SIGCSE Bulletin*, volume 37, pages 186–190. ACM.

Burmistrov, I. and Leonova, A. (2003). Do interrupted users work faster or slower? the microanalysis of computerized text editing task. *Human-Computer Interaction: Theory and Practice (Part I)–Proceedings of HCI International*, 1:621–625.

Card, S. K., Moran, T. P., and Newell, A. (1980). The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, 23(7):396–410.

Carter, A. and Sangler, C. (1997). *The Programmers Stone*. Available at https://www.datapacrat.com/Opinion/Reciprocality/r0/index.html.

D'Ambros, M., Lanza, M., and Robbes, R. (2010). An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE.

Ducasse, S., Anquetil, N., Bhatti, M. U., Hora, A. C., Laval, J., and Girba, T. (2011). Mse and famix 3.0: an interexchange format and source code model family.

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society.

Hassan, A. E. and Holt, R. C. (2003). The chaos of software development. In *null*, page 84. IEEE.

Hou, D., Jablonski, P., and Jacob, F. (2009). Cnp: Towards an environment for the proactive management of copy-and-paste programming. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 238–242. IEEE.

Hunt, A. and Thomas, D. (1999). *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Jeffries, R., Turner, A. A., Polson, P. G., and Atwood, M. E. (1981). The processes involved in designing software. *Cognitive skills and their acquisition*, 255:283.

Kamma, D. (2014). Study of task processes for improving programmer productivity. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 702–705. ACM.

Kim, M., Bergman, L., Lau, T., and Notkin, D. (2004). An ethnographic study of copy and paste programming practices in oopl. In *null*, pages 83–92. IEEE.

Lehnert, S. (2011). A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 41–50. ACM.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition.

McConnell, S. (2004). *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA.

OMG (2008). Software & systems process engineering metamodel specification (v2.0).

Polson, P. G. and Kieras, D. E. (1985). A quantitative model of the learning and performance of text editing knowledge. *ACM SIGCHI Bulletin*, 16(4):207–212.

Rahman, F. and Devanbu, P. (2013). How, and why, process metrics are better. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 432–441. IEEE.

Rosson, M. B. (1983). Patterns of experience in text editing. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 171–175. ACM.

Rubin, V., Lomazova, I., and van der Aalst, W. M. (2014). Agile development with software process mining. In *Proceedings of the 2014 international conference on software and system process*, pages 70–74. ACM.