When Do You Repeat Yourself? Voices from the Trenches of Linux Kernel Maintainers on Code Duplication

Anonymous Authors

***Abstract*—The vast scale and continuous evolution of the Linux kernel make its maintenance a complex undertaking, where code duplication remains a persistent challenge that can hinder development and introduce bugs. This paper presents a multimethod ethnographic study investigating the socio-technical dynamics of contributing to the Linux kernel by reducing identified code duplication. To support this study, we developed a command-line utility tool for detecting function-level duplications, offering a concrete entry point for new contributors. Our ethnographic approach, including complete participant observation and participant-as-observer analyses, demonstrated that addressing duplications is viable for lowering the contribution barrier. This point is evidenced by 8 of 13 (62%) accepted contributions (patches) from 24 newcomers (16 undergraduate and 8 graduate students) in the Linux kernel project, collectively removing 1.397 lines of duplicated code. Nevertheless, the study reveals a more intriguing and complex reality beyond mindlessly removing clones. An analysis of maintainer feedback on both accepted and rejected contributions highlights a nuanced understanding of technical debt due to code duplication within the Linux kernel community, where the benefits of eliminating duplications are carefully weighed against factors such as readability, the introduction of new abstractions, and the specific code context.**

I. Introduction

Hundreds or thousands of developers working together on a software product is a complex and demanding task. Similar or even exact copies are a frequent and often unavoidable outcome. Excessive duplication can negatively impact the maintainability of a project, as, when it occurs, any change to a repeated code segment (to fix a bug, for instance) requires replication across all other copies. Developers must, therefore, remain constantly vigilant to synchronize different parts of the codebase, which is a highly error-prone yet seemingly simple responsibility. Regarded as a major source of technical debt, removing or refactoring duplicated code demands great care to preserve the original behavior, with minor mistakes possibly leading to serious problems in the future.

A widespread dogma among developers is the *Don't Repeat Yourself* principle [1], stated as follows.

> **DRY principle:** *"We feel that the only way to develop software reliably, and to make our developments easier to understand and maintain, is to follow what we call the DRY principle: Every piece of knowledge must have a single, unambiguous, authoritative representation within a system".*

The Linux kernel is a foundational Free/Libre and Open Source Software (FLOSS) project, a vital component of the world's digital infrastructure. Considering only lines of code (no comments or blank lines) of strictly programming languages, in version 6.17, the main repository has more than 27 million lines of code and involved contributions from over twenty thousand developers. Within this context, code duplication remains a significant challenge, reducing the code readability and increasing the risk of introducing bugs [2], [3]. This issue is particularly prominent in kernel device drivers, which account for over 66% of the Linux kernel source code.

The detection of code duplication, or *code clones*, has been a research topic for decades [4]. The literature offers a widely accepted taxonomy that classifies clones into four types based on their similarity, from exact copies (Type-1) to semantically equivalent but syntactically different fragments (Type-4) [5]. Several detection techniques have emerged, including textual, token-based, tree-based, and graph-based approaches [5]. These efforts have culminated in state-of-the-art techniques, such as the graph-based approach proposed by Liu et al. [6].

To address code duplication in the Linux kernel, we developed a command-line tool capable of detecting and analyzing function-level duplications[1]. We conducted a multimethod ethnographic study leveraging the tool capabilities to identify duplications and submit contributions, directly or indirectly, to the Linux kernel, aiming to remove redundant code. Through interactions with maintainers, this study gained insights into how the Linux community perceives the code quality related to code clones. Contrary to the DRY principle, our findings indicate that, in some contexts, maintainers can tolerate duplications for better readability, performance increases, or even to avoid integration overhead.

Ethnography is a well-established qualitative method for understanding people, their cultures, and work practices [7]. It provides insights into community members' values, beliefs, and practices [8]. In this study, we first employed a complete participant observation approach [9], actively contributing by performing deduplications while documenting the process. We then conducted a participant-as-observer study [9] with students resolving duplications identified by the tool. Beyond demonstrating that the tool lowered the entry barrier for new contributors, these studies yielded intriguing observations on kernel Linux deduplications. This paper describes the afore-

[1]The tool and its documentation will be referenced after the review process.

mentioned studies, along with our visions and reflections from diving into the Linux kernel development process.

II. The Linux Kernel

The Linux kernel is a complex project organized into subsystems, such as the process scheduler, memory management, and device drivers. Each subsystem typically has a dedicated maintainer or a team of maintainers overseeing its development and managing contributions. The development process is coordinated through Git (also designed by Linus Torvald, focused on the Linux kernel development), and contributions are formatted as patches, text documents that outline the differences between two source code versions. These patches are then submitted to the relevant Linux mailing lists for public review and discussion.

This study focuses on two of these subsystems, selected to represent different contexts for contribution. We explore the *AMD Display drivers*, chosen for their importance within the hardware ecosystem, which is responsible for natively enabling AMD GPU functionality in Linux. This subsystem is substantial, with over 391,000 lines of code across more than 1,089 files. We also investigate the *Industrial Input/Output (IIO)* subsystem, often considered an accessible entry point for new contributors. The IIO subsystem provides support for devices that perform analog-to-digital or digital-to-analog conversions[2] and contains over 281,772 lines of code in more than 755 files.

III. Research Strategy

We employed a multi-method research approach. We developed and evaluated a proposed tool using a method from the literature [10]. Given this validation, we proceeded to understand the viability of reducing duplications found by the tool with an ethnographic study using a participant observation methodology [9]. In the first phase, we interacted with the Linux kernel community and got their feedback (*complete participant observation* [9]) through sending deduplication patches to the AMD Display driver. In the second phase, during a university course, we proposed a similar activity of sending patches to AMD Display or IIO to students. At the same time, we guided and mentored them throughout the process (*participant-as-observer* [9]).

A. Tool

Academic research on code duplication often focuses on pairwise comparison of code artifacts, which is not directly applicable to comprehensive codebase analyses. Existing free software tools we explored in practice suffered from limited functionality [omitted]. Thus, we developed a command-line tool[3] designed to detect and analyze function-level code duplication within large codebases, such as the Linux kernel. The tool is released under LGPLv3 (GNU Lesser General Public License, version 3) and employs a two-stage architecture that separates the computationally intensive task of finding duplicates from the process of querying the results. The first stage analyzes the entire codebase, parses C functions, and stores identified duplicates in a database. The second stage uses this database to retrieve information about the identified duplications quickly.

The tool was designed to address the specific challenges posed by large-scale codebases, focusing on scalability and prioritization of duplications that may impact maintainability. It uses a text similarity method based on TF-IDF (Term Frequency-Inverse Document Frequency) vector embedding, implemented with the Gensim library [11], and computes cosine similarity to detect clones.

We compared our tool against the BigCloneBench dataset [10]. This validation showed that the tool is capable of detecting the majority of Type-1 and Type-2 duplications (100% of Type-1 and 85% of Type-2 with a cosine similarity threshold of 0.9), demonstrating sufficient accuracy to support our ethnographic studies.

B. Complete Participant Observation Study

To investigate whether we could use the tool to find ways to contribute to the Linux kernel, we first conducted a complete participant observation experiment, where the main author (a graduate student) acted as a first-time contributor to the AMD Display driver, collecting artifacts of our experience in the process.

We executed the tool in the AMD Display driver codebase. We manually analyzed the biggest duplications regarding the number of lines, finding one pair that we judged promising to try to deduplicate. We approached big duplications as proof of concept to gauge the tool's potential for enabling significant impacts.

Given the duplicated function pair, we observed that the files in the functions contained multiple other duplications. Thus, we proposed a simple systematic approach to mitigate all the functions duplicated in the context, not just the duplicated function pairs. After approaching the duplications with a systematic strategy, we sent a patch to the AMD development mailing list for the feedback of maintainers, while documenting the process, interactions, and impressions (available in our Replication Package).

In C programming, communication between source files is achieved by creating header files that specify libraries [12]. Since the Linux kernel is primarily written in C, our systematic strategy consisted of eliminating code duplications through consolidating the duplicated code into a single library, which would replace instances of duplication across the codebase. For each duplicated function pair, for each function in the pair, we used the tool to locate all duplications of the function in the codebase, as it could have been duplicated more times than the two occurrences detected initially. This strategy resulted in a collection of duplicated functions and corresponding code files.

To identify shared code more effectively, we extended this approach to search for other common functions across all collection files. We then applied specific refactoring methods

[2]https://kernel.org/doc/html/v6.17/driver-api/iio/intro.html

[3]Anonymous repository: https://anonymous.4open.science/r/arkanjo-C548

to each shared function. If the functions were identical across files, we removed the duplicates and created a single function in the library. If modifications existed, we applied case-specific refactoring.

C. Participant-as-Observer Observation Study

To broaden our study and observations, we adopted a participant-as-observer approach, involving students as free software project newcomers to make practical contributions to the Linux kernel. This approach formed the core activity of a university course, in which we guided students to remove code duplications in the Linux kernel as one possible pathway for contributing to the project in the first phase of the course [omitted].

The 2025 course offering had 37 students (25 undergraduate and 12 graduate), who were asked to form groups of two or three members, but graduate students could work individually. To assist students in this task, we (a professor and three more experienced graduate students) prepared several alternatives for contributing to the Linux kernel, including simpler contributions such as fixing code style issues. In this context, we specified two options for removing duplications in the Linux kernel.

As the first option, we ran the tool on the IIO subsystem, creating a list of duplicated function pairs. Then, we curated the list even more to get more actionable duplications for the students using this approach:

1) We filtered the list to keep only duplicated pairs that happened in the same file, because the IIO subsystem is very decentralized, and a duplication between two different drivers may involve the cooperation of different maintainers and/or the unification of totally different interfaces;
2) We removed duplications that were too short and would not represent a significant contribution;
3) We manually ranked code duplications and marked some duplications with insights based on personal opinion and perceptions over the remaining entries. This was done to guide students so that they could approach the technical debt with a greater chance of having their contribution merged into the code.

The students' task was to select a recommended entry from the list, refactor the code to eliminate these duplications, and then submit their patches to the IIO mailing list for the maintainers' review.

For the second alternative, students were offered an experience similar to our initial complete participant observation. In this approach, students were responsible for the entire workflow: executing the tool, independently analyzing the results to identify a potential deduplication, creating a patch, and sending the patch to the driver's maintainers.

Of the students who chose to work on the tool-related tasks, 23 (16 undergraduate and 7 graduate) formed 11 groups to pursue the first alternative. One graduate student opted for the second alternative. Notably, none of these students had prior experience contributing to the Linux kernel, making this their first attempt to submit a patch as newcomers to the project.

Using blog posts, the groups documented their experiences and approaches to refactoring the duplications, as well as their experiences submitting a patch to the Linux kernel [omitted]. We also analyzed these posts to understand the refactoring patterns used in deduplications, the experience of sending patches, and the opinions of subsystem maintainers on them.

IV. Results

In the **complete participant observation** study (first phase), we chose one function pair found on the AMD Display Driver to approach and apply the systematic approach. We reached a point where we had an idea of removing the duplications found, but we could not complete the refactoring to a state capable of submitting as a patch to the driver. We found that the configuration files on the AMD Display driver are designed to be imported at compilation time using *#define* macros. The configuration files' design choice makes refactoring duplications on generic approaches trickier, as refactoring code that depends on these files requires significant modification of the configuration files' design and deep knowledge of the codebase, which we, as first-time contributors, did not have. Thus, we opted not to continue investigating this deduplication. We found that the duplicated function from the code reduction existed in five code files, and two other functions were duplicated in those five. All functions across the files were exactly equal, so there was no need to apply the refactoring methods presented in the literature. The refactoring was resumed to create a generic library and fix the compilation targets.

In this attempt, we reached a point where the deduplication was good enough to submit it as a contribution to the AMD Display driver code quality. Thus, we moved to sending the refactoring to the driver maintainers as a patch.

In the feedback on the first submitted version, the maintainers only asked for minor changes to align with the coding style, correct license use, and align with initially unknown conventions. We submitted a second version, in which the maintainers asked us to move the functions created in the generic library to an existing file instead of creating a new one. We sent a third version, and the patch was accepted and integrated into the kernel codebase on February 25, 2025.

Since we suffered a delay in the review process, we directly contacted a maintainer to understand why. From this conversation, we understand that one reason for the delay was that duplicated code enhances the independence of GPU driver code. This approach allows developers to make changes to a specific GPU without needing to test compatibility with others, which helps save a significant amount of time and effort. This example of hardware variations, also mentioned by Kapser and Godfrey [13], leads us to the primary conclusion of our complete participant observation:

Takeaway T1: *Driver developers do not necessarily view duplicated code negatively, which contradicts standard software engineering practices.*

Building on this insight, we realized that the perception identified in T1 might not be limited to driver developers alone. To deepen our understanding, we systematically examined other contributions related to code duplication reduction and analyzed how different maintainers responded to such patches. This broader analysis aimed to capture diverse maintainer perspectives and contextual factors influencing their decisions.

Table I summarizes all the contributions sent by the main author and student groups (GID) [4]. It contains the information about the chosen subsystem (SS) and the similarity threshold (SML). Each entry also has the patch status, takeaways that may have surfaced from the interaction, and a code differential corresponding to the final patch versions before it was dropped or accepted into the code base. The code differential for a set of patches corresponds to the sum of added and removed lines of each patch.

TABLE I
SUMMARY OF NEWCOMER DEDUPLICATION CONTRIBUTIONS.

GID	SS	SML	Patch Status	Takeaway	Diff
0	AMD	100%	Accepted (v3)	T1	+114/-520
1	IIO	100%	Dropped (v2)	T2, T3	+7/-14
2	IIO	100%	Dropped (v1)	T2	+23/-27
3	IIO	100%	Accepted (v1)	None	+12/-30
4	IIO	100%	Accepted (v3)	T2	+2/-7
5	IIO	90%	Accepted (v1)	None	+17/-70
6	IIO	90%	Dropped (v1)	T4	+95/-92
7	IIO	90%	Accepted (v3)	None	+20/-36
8	IIO	90%	Accepted (v4)	T5	+20/-16
9	IIO	90%	Accepted (v6)	T5	+149/-243
10	IIO	90%	Dropped (v1)	T2, T5	+32/-58
11	IIO	100%	Dropped (v1)	None	+2/-12
	AMD	90%	Accepted (v2)	None	+90/-489

Regarding our analysis from the **participant-as-observer** phase, we identified that most groups applied the parameterized refactor method, while a few used the extractor method. For instance, groups 1 and 4 worked on similar duplications, where two functions were involved: one to determine if a register was volatile and another to determine if it was writable. These two functions returned opposite Boolean results. Group 1 initially addressed the problem using the extract method, while Group 2 used macros. After feedback from maintainers, both solutions converged toward keeping one of the functions while modifying the other to return a negated query of the former function.

By the end of the feedback cycle, the patch from Group 4 was accepted, but the final implementation differed substantially from the original. The change was primarily driven by a maintainer's feedback: *"I think the old code is more readable than hiding the values in a macro even if it is duplicating a few lines of code."* This response category revealed a recurring concern among maintainers that went beyond functional correctness or redundancy reduction. Instead, it reflected a preference for code clarity and comprehension. Building on these consistent insights from multiple review interactions, we formulated our second takeaway:

Takeaway T2 (Readability): *Maintainers prioritized readability over removing duplications in many different contexts. In their view, going against the* ***DRY principle*** *was worth it if it resulted in easier-to-read and understand code.*

Maintainers considered the patch from Group 1 irrelevant. The reason is that Group 1 chose to apply the refactoring to a device that contained a register that was both read-only and non-volatile. Thus, the technical debt selected by Group 1 was not merely a duplication but an incorrect implementation of the method.

Quote: "[...] such patches might not worth it since the proposed improvement is very small (and questionable) while the upstreaming process still requires some effort."

Takeaway T3 (Integration Overhead): *A maintainer suggested that the effort put into the contribution after it was accepted (i.e., the process of propagating the change to upper maintainers) was considered. In other words, maintainers may* ***evaluate the trade-off between the impact of contributions (not only deduplications) and the struggle to integrate them into the complex Linux project****.*

Groups 3, 6, and 10 created a struct to encapsulate the parameters and remove the duplication by applying the parameterized method. The patches from Groups 6 and 10 were rejected by the same maintainer who approved the patch from Group 3, tackling the duplication of a simple iterative algorithm, which also contained a bug that could be inferred by inspecting both functions. In this case, the group refactored code from a more critical technical debt and took the opportunity to fix an unnoticed incorrect behavior.

The maintainer's justification for rejecting the patch from Group 6 was that it proposed to fix a duplication while adding more lines than removing, and that it was not convincing enough in other aspects, such as readability.

Takeaway T4 (Cognitive Load): *Needing to jump to a function definition (to see its behavior) or a function call (to see its concrete parameters) in order to understand code is a cognitively demanding task for developers [14]. Duplications that avoided this were justifiable to reduce cognitive load when parsing through the code.*

Quote: "Wrapping this up doesn't provide any real advantage, requiring as it does the reviewer to look at this function AND where the value is set rather than seeing them in one place."

[4]As in the first phase, the first author is also a newcomer, we included this experience in the table as Group 0. Nevertheless, the subsequent analysis will focus more on the patches sent by students.

The reason for rejecting the patch from Group 10 was that they refactored code files with simple context; therefore, adding a layer of abstraction to remove duplication decreased code readability without clear gains in codebase quality. Interestingly, both rejected groups addressed duplications across initialization functions for different devices, where code readability was vulnerable to refactoring methods (T2 + T5). [blabla] Group 9 initially approached the selected duplication with a parameterized method, but after entering the feedback cycle, it was revealed that the problem represented a much more complex technical debt. From the first version to the last (version six), the patch was split into a series with four patches, where two of them have been applied, and the remaining entries are receiving suggestions. Even though the patch became a series of four, only one of them (one of the accepted) was strictly tackling duplication.

Quote: "Note that is not an appropriate change for the large reads though as `spi_write_then_read()` bounces all buffers and so would add a copy to those high(ish) performance paths."

Takeaway T5 (Performance): *Given the low-level development context of Linux and the IIO subsystem, which is usually related to devices with limited resources, we detected hints that duplication may be forgivable if it improves performance.*

Groups 7 and 8 used the extract method to approach the deduplications. Group 8 sent their patch to the maintainers, and the code changes were approved, with minor requested changes in code style and smaller errors fixed (++T5). In the case of Group 7, the maintainer initially rejected the proposed patch but suggested a new refactoring approach: applying the extraction method refactoring to the duplicated code to an existing helper function. After the feedback cycle, maintainers accepted the patch in version 3.

Group 11 was a single student who took the second option and executed the tool, completing the entire process of identifying duplications himself. The student identified a duplication in the driver that occurred across eighteen code files, creating a patch to remove approximately four hundred lines of duplicated code. The student sent the deduplications to the maintainers, and their feedback pointed to fixing warnings in the code without arguing about the merit of the duplication removals. After addressing the issue, the patch was applied. This student also contributed to IIO, tackling a code duplication with inline method refactoring. This one, however, was received with more skepticism by the maintainer, and the student dropped the contribution because he did not think the feedback instructions were clear enough.

Analyzing the students' approaches to the duplications and the maintainers' feedback, we saw that people without previous experience in Linux kernel development could approach the duplications found by the tool as a comprehensive and more straightforward way to become contributors to the kernel. We observed that not all duplications are viewed as code of bad quality, with maintainers analyzing the trade-off between the purpose of the duplicated code and many other factors.

The eleven groups, which have submitted 11 contributions to the IIO subsystem, have collectively requested the insertion of 379 lines and the deletion of 605 lines in the subsystem, representing an average of 34.45 lines added and 55 lines removed per contribution.

Considering the sum of fully merged contributions, we have a code diff value of +51/-143, with an average of +12.75/-35.75. This ratio is better than dropped contributions, which have a total of +179/-219 with an average of +29.83/-36.5.

It is also possible to notice a difference based on the similarity threshold used to detect duplications. The proposed patches to address 100%-similarity duplications summed up to +46/-90 (average of +9.2/-18). For a 90% value, the result was a total of +333/-515 with an average of +55.5/-85.83. The 90% value enabled the tool to identify less strict duplication cases, and, therefore, code segments with a greater extension, compared to the 100% similarity entries, which were shorter segments, causing a difference in the average number of modified lines.

Finally, the AMD contribution from Group 11, which involved the complete use of our developed tool components, got a completely different outcome. With a total of 90 lines inserted and 489 lines removed, there are some points to be considered for the success and high modification extension of the contribution: (1) the AMD Display subsystem has some self-reported problems regarding overall technical debt; (2) the AMD Display subsystem has a more centralized code structure and maintainer hierarchy, enabling contributors to tackle code duplications across a broader scope and multiple files, which happened to Group 11's contribution;

We derived four takeaways from **participant-as-observer observation**. These takeaways are accompanied by example quotes from the maintainers that originated during the review process [5].

V. Concluding Remarks

Ethnographic studies, including full participant and participant-as-observer observations, provided a realistic view of the opportunities and challenges in removing code duplications detected by the tool. First-time Linux kernel contributors could effectively use the tool to identify duplications and submit patches.

While this research included one successfully merged patch by the authors, and 7 out of 12 patches sent by students were accepted and merged into Linux, a portion of the students' efforts encountered hurdles: 4 student groups had their patches rejected.

Nevertheless, our work suggests that maintainers do not always view code duplication as inherently unfavorable, especially when it improves clarity, performance, and more. This suggests a nuanced understanding of code quality within the

[5]It is paramount to state that we **are not affirming** that these quotes represent the complete positioning of the maintainers who wrote them on code duplication.

Linux kernel community that can go against the DRY principle, where practicality and maintainability in specific contexts sometimes outweigh the general principle of eliminating all code duplication.

Replication Package

All anonymized threads containing the entire review process and interactions with maintainers are available in this Zenodo repository: https://doi.org/10.5281/zenodo.17432193.

To avoid violating the double-anonymized review process, the anonymous repository of the developed tool is available at https://anonymous.4open.science/r/arkanjo-C548. The tool website link and omitted references will be added in the camera-ready version of the paper.

References

[1] A. Hunt and D. Thomas, *The pragmatic programmer: from journeyman to master*. USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[2] W. Hordijk, M. L. Ponisio, and R. Wieringa, "Harmfulness of code duplication-a structured review of the evidence," in *13th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. BCS Learning & Development, 2009.

[3] K. Hotta, Y. Sasaki, Y. Sano, Y. Higo, and S. Kusumoto, "An empirical study on the impact of duplicate code," *Adv. Soft. Eng.*, vol. 2012, 1 2012. [Online]. Available: https://doi.org/10.1155/2012/938296

[4] H. T. Jankowitz, "Detecting plagiarism in student pascale programs," *The computer journal*, vol. 31, no. 1, pp. 1–8, 1988.

[5] C.-F. Chen, A. Zain, and K.-Q. Zhou, "Definition, approaches, and analysis of code duplication detection (2006–2020): a critical review," *Neural Computing and Applications*, vol. 34, pp. 1–31, 08 2022.

[6] J. Liu, J. Zeng, X. Wang, and Z. Liang, "Learning graph-based code representations for source-level functional similarity detection," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 345–357.

[7] W. A. Edmonds and T. D. Kennedy, *An applied guide to research designs: Quantitative, qualitative, and mixed methods*. Sage Publications, 2016.

[8] H. Sharp, Y. Dittrich, and C. R. De Souza, "The role of ethnographic studies in empirical software engineering," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 786–804, 2016.

[9] R. L. Gold, "Roles in sociological field observation," *Social Forces*, vol. 36, pp. 217–223, 1958.

[10] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 476–480.

[11] R. Řehůřek, P. Sojka *et al.*, "Gensim—statistical semantics in python," *Retrieved from genism. org*, 2011.

[12] B. W. Kernighan and D. M. Ritchie, *The C programming language*. prentice-Hall, 1988.

[13] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful," in *2006 13th Working Conference on Reverse Engineering*, 2006, pp. 19–28.

[14] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: Association for Computing Machinery, 2006, p. 23–34. [Online]. Available: https://doi.org/10.1145/1181775.1181779