Empirical Investigation of Code Quality Rule Violations in HPC Applications

Shahid Hussain
shussain@uoregon.edu
Department of Computer and Information Science
1202 University of Oregon
Eugene, Oregon, USA

Kaley Chicoine
kmchicoine@gmail.com
Department of Computer and Information Science
1202 University of Oregon
Eugene, Oregon, USA

Boyana Norris
norris@cs.uoregon.edu
Department of Computer and Information Science
1202 University of Oregon
Eugene, Oregon, USA

ABSTRACT

In large, collaborative open-source projects, developers must follow good coding standards to ensure the quality and sustainability of the resulting software. This is especially a challenge in high-performance computing projects, which admit a diverse set of contributions over decades of development. Some successful projects, such as the Portable, Extensible Toolkit for Scientific Computation (PETSc), have created comprehensive developer documentation, including specific code quality rules, which should be followed by contributors. However, none of the widely used and highly active open-source HPC projects have a way to automatically check whether these rules, typically expressed informally in English, are being violated. Hence, compliance checking is labor-intensive and difficult to ensure. To address this issue, we propose an automated method for detecting rule violations in HPC applications based on the PETSc development rules. In our empirical study, we consider 46 PETSc-based applications and assess the violations of two C-usage rules. The experimental results demonstrate the efficacy of the proposed method in identifying PETSc rule violations, which can be broadened to other HPC frameworks and extended by us and others in the community to include more rules.

CCS CONCEPTS

• **Software and its engineering** → **Abstraction, modeling and modularity**.

KEYWORDS

Code quality, high-performance applications, developer rules violations, process improvement, PETSc, libclang, LLVM

ACM Reference Format:
Shahid Hussain, Kaley Chicoine, and Boyana Norris. 2021. Empirical Investigation of Code Quality Rule Violations in HPC Applications. In *Evaluation and Assessment in Software Engineering (EASE 2021), June 21–23, 2021, Trondheim, Norway*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3463274.3463787

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EASE 2021, June 21–23, 2021, Trondheim, Norway
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9053-8/21/06…$15.00
https://doi.org/10.1145/3463274.3463787

1 INTRODUCTION

Large-scale open-source projects with many diverse contributors experience unique software quality challenges, which can impact the sustainability and developer productivity. There are several factors which are either directly related to software quality, such as number of defects, or indirectly related, such as maintainability. The indirect software factors depend on the content and quality control of activities which are performed during the development process. The source code analysis is considered as an important activity of the application development process to ensure product quality. In contrast, manual code review is less scalable and consumes significantly more time and effort. Automated source code analysis can help tackle some of these challenges [22]. Well-defined programming styles [6], and coding standards [5, 29] are often used to ensure code quality and improve software maintainability and developer productivity. In industry, project managers can ensure that there is sufficient focus on code quality in their teams, because normally developers may sidestep quality standards (especially coding standards) to meet the deadlines. In the open-source development community, and in the high-performance computing sub-field in particular, the developers are often a loosely coupled collection of multidisciplinary students, postdocs, scientists and programmers, and for most of them, software development is not the only or the primary job responsibility. Moreover, many do not have any formal computer science training. Hence, it is critical to enable the defined rules and automatic detection of rule violations in HPC libraries and applications.

High-performance computing used to be a niche computational area of very large-scale supercomputers and complex scientific simulations. However, over the last decade, the re-emergence of AI and large-scale data science has broadened its importance – most modern machine learning methods rely on scalable, fast numerical libraries, such as the ones we consider in this study. Several software libraries, including hypre [16], Trillions [28], SuperLU [27], and PETSc [1, 3] have been used to develop robust, scalable, and efficient applications in many domains, including AI, data science, medicine, physics, chemistry, and biology. The main advantage of using such libraries is not just to reduce application coding effort, but to leverage advanced mathematical approaches and scalable implementations developed by thousands of experts over several decades [2, 10].

The Portable, Extensible Toolkit for Scientific Computation (PETSc) is one of the widely used software libraries for solving problems related to partial differential equations using C, C++, and Fortran for large scale parallel numerical computations. PETSc has been actively developed for over 25 years, with 46 current active developers and hundreds of contributors over its existence. At the time of this article, the code size is 800,000 lines, with 82% ANSI-C code. PETSc provides a suite of components for the scalable (hundreds of thousands of tasks on distributed-memory parallel resources) solution of complex problems, including nonlinear partial differential equations, sparse and dense linear systems, nonlinear optimizations, and other critical numerical building blocks.

Several additional toolkits and libraries have been developed on top of PETSc, and use its development guidelines [21], e.g., ADflow, DEAL.II, MFEM, OpenFOAM, libMesh, and MOOSE. Moreover, PETSc includes many example applications that solve partial differential equations and other numerical problems, such as `ex5.c` (in `src/snes/tutorials/`), which implements the "Bratu nonlinear method to solve SFI (Solid Fuel Ignition) problem in 2-D the rectangular domain." Over the last decades, numerous large-scale HPC applications have been developed using these HPC libraries in areas including aerodynamics, cancer surgery, computational fluid dynamics, data mining, seismology, and many others. Therefore, it is imperative to ensure the consistency of quality and development growth of these important libraries and their applications. To our knowledge, there is no approach that enables the definition of custom code quality rules by developers and automates the detection of rule violations accurately and efficiently. Consequently, in this study, we demonstrate to the SE research community the types of rules large-scale HPC projects include, and an automated violation detection approach that can be used throughout the development of an HPC application. We present results from applying this method to a set of test applications.

For this study, we consider the PETSc release (i.e., petsc-3.14.3) which was released on Jan. 10, 2021. We analyze 46 applications that solve complex mathematical problems in various scientific areas, including solving the Bratu equations for a solid fuel ignition problem, modeling fluid flow through a driven cavity, nonlinear elastic problems, such as incompressible Neo-Hookean solid modeling, and others. The PETSc rules documentation for the application developers is divided into three sections, **Naming**, **C-Usage**, and **PETSc functions and macros** [21]. In this empirical study, we consider the two C-Usage rules shown below.

Rule-1: Do not use `if (rank==0)` or `if (v==NULL)` or `if (fig==PETSC_TRUE)` or `if (fig == PETSC_FALSE)`. Instead, use `if (!rank)` or `if (!v)` or `if (fig)` or `if (!fig)`.

Rule-2: Do not use `#ifdef` or `#ifndef`. Rather, use `#if defined(...` or `#if !defined(...`. Better, use `PetscDefined()`.

The aim of the proposed methodology is to automatically discover PETSc code quality rule violations (such as the two rules shown above) from HPC applications developed using the constructs of C/C+ language. The proposed methodology is implemented by using libclang (a library of Clang) for parsing C/C++ applications and LLVM (Low Level Virtual Machine), a collection of modular and reusable compiler and toolchain technologies [19]. We apply these two rules to 46 applications developed by the core PETSc team and contributed by external developers. We perform several experiments and report the results with respect to construct frequency and rule violation severity for each application. The experimental results demonstrate the efficacy of the proposed method and motivate further work on expanding the automation to more project-specific rules and analyzing a broader set of codes.

2 RELATED WORK

The Software Engineering (SE) research community has considered several factors that can affect the quality of open-source software development and sustainability, such as employment of design patterns and code refactoring [14, 15], and implications of coding standards [5, 11, 21, 24]. Even though these factors help software teams ensure code quality, they also increases the effort required for code reviews[4]. In this regard, several tools have been introduced for automating code review[9], dead code prediction [8, 9, 17], and code debloating [18, 23], and static and dynamic analysis of source code [12, 13, 20, 26], aiming either to help in code refactoring or to identify unnecessary code.

The SE literature includes reports of several coding standards and their positive impact on the software quality such as Misra-C [20] target the JSF Air Vehicle standard for the safe side use of C language in the development of applications for the critical systems. Similarly, Holzman [13] presents a set of rules to design and implement safety-critical software. Similarly, Balay et al. [21] introduce a set of rules to design and implement HPC applications using C/C++ language, grouping them into several categories, including naming, style, and C usage. In our study, we empirically investigate the C-usage related developer's rules violation in PETSc-based applications.

Baum et al. [4] performed a qualitative study and gathered the responses of 240 developers to assess and report the current state of art practices used for code review. The authors have reported the change based and modern code review practices through a statistical analysis. There are several tools which help to detect the dead (unnecessary) code such as IntelliJ IDEA [9] and Android Studio [26] to detect dead code by using runtime information and dependencies at class level. Similarly, in their study Eichberg et al. [8] introduce a new static approach for the identification of infeasible paths in source code at an abstract level to detect bugs.

In their study Jiang et al. [17] introduce a tool entitled JRed for static analysis of Java applications at class and method levels for reduction in application size and surface attacks. JRed builds a call graph for each Java-based application and its libraries and trim the unreachable code. Subsequently, in their study, Sharif et al. [23] introduce a new tool entitled TRIMMER, which debloats applications that are compiled through LLVM. The aim of TRIMMER is to prune code until it contains no unreachable code.

Juergens et al. [18] perform an empirical study to suggest a feature profile to control the usage of a system at application level. The authors found that 28% features were not used, which indicates that extensive amount of code may be acceptable to remove. Similarly, Eder et al. [7] gather data of two years for a business information system at the method level and report that 25% of methods were never used and maintenance efforts could be reduced.

Sora [25] statically analyzes the dependency structure of the system by considering the code's centrality and identified the code as necessary of unnecessary code of the system. Subsequently, Haas et al. [12] proposed a methodology to present recommendations

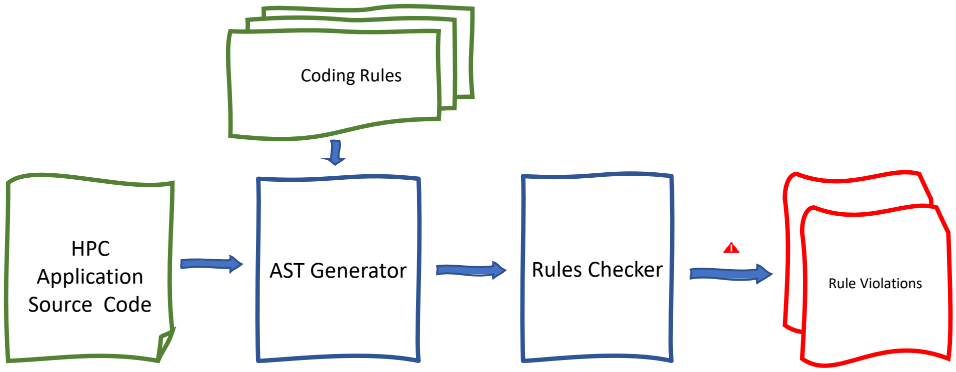

Figure 1: Workflow of the proposed method

for unnecessary code through static analysis at file level and determine the set of stable and least central files. The authors identified stability through a set of metrics.

3 PROPOSED METHOD

In this study, we propose a method to assess the PETSc developer rule violations in the development of HPC applications using C language constructs. The layout of the proposed method is shown in Figure 1. The descriptions of the constituent components follow in the rest of this section.

3.1 HPC Applications

In this study, we consider PETSc v. 3.14.3 released on Jan 10, 2021. To assess developer rule violations, we analyze the 46 PETSc examples included the SNES component implementation (Newton-based nonlinear partial differential equations solution methods). Short descriptions (from comments) and statistics for each application are shown in Table **??** of Appendix A. Most application names start with the label "ex" and followed by a number, such as `ex5.c`. Similarly, we also include applications with the same name, but representing distinct codes in different subcomponents, such as application `ex1.c` exist in two different subdirectories, namely `tutorials` and `network`.

3.2 AST Generator

The AST generator component of the proposed method is implemented using the libclang library (part of the widely used Clang C/C++ frontend of the open-source LLVM compiler framework). We chose libclang/LLVM because it not only enables parsing of C/C++ code through simple interfaces, but also provides rich data structures and algorithms for analyzing various levels of the internal code representation, such as function calls, arguments, types, etc. Several other static checking tools successfully leverage the same infrastructure.

3.3 Rule Checker

The aim of this part of the proposed method is to take AST as input and traverse it according to the description of an implementing rule. Moreover, it could be tuned according to description of rules. Since we are considering two PETSc rules related to C-usage. Consequently, we design and implement Rule Checker accordingly. Note that the implementation of each rule is short, using the Python libclang interface. Our long-term goal is to create a high-level interface that further simplifies rule definition, so that the core project developers can easily add new custom rules, specific to their project. Finally, the traversed information is shown in a dataset structure shown in Table1.

3.4 Rule violations

The aim of this part is to report the PETSc developer's rule violations which could aid developers to analyze and benchmark the quality of an application. Minimizing false positives is critical because a high false positive rate would decrease developer productivity, or worse, cause them to not use this analysis at all.

4 RESULTS AND DISCUSSION

We applied our tool to identify Rule-1 and Rule-2 violations in 46 PETSc applications. The detailed results for rule occurrences and violations are included in Table**??** of Appendix A. In summary, the main observations based on these experimental results are as follows.

- We found rule violations in half of the applications.
- To validate these results and investigate false positive cases, we performed careful manual analysis of each application. For example, in the case of application ex3.c, we observed the use of if-construct in 18 places, with two of these occurrences detected as Rule-1 violations, as shown in Table 2.
- Most Rule-1 violations are observed in the same function, such as in case of ex3.c where both violations occur in the "FormJacobian" function. In all PETSc applications, this function, and all functions it calls comprise the bulk of the application-specific functionality and is invoked multiple times by the nonlinear solver components in PETSc.

Figures 2 and 3 show the frequency analysis and rule violation severity defined as the ratio of violations and total construct occurrences (shown as a percentage). In Figure 2, we observe a maximum of 17 Rule-1 violations in the ex70.c application (Poiseuille flow problem), and the fewest number of Rule-1 violation in the ex30.c application (steady-state 2D subduction flow, pressure, and temperature solver). In Figure 3, we observe the highest Rule-1 violation severity in ex25.c application rather than `ex70.c`, which is due to its much larger actual number of construct occurrences. We observe

Table 1: Overview of Dataset structure

Variable	Description
Path	The path of the top-level directory of the application.
Directory_Name	The (relative) directory name of a traversed application.
File_name	Name of the file containing the "main" function for the application.
Functions	Total number of user-defined functions in the application.
Total rule constructs	Total number of rule-related constructs in the application, e.g., for Rule-1 it contains the total number of if-statement occurrences, and for Rule-2, it contains the total number of macros.
Rule violations	Total number of rule violations for a specific application.

Table 2: Example of a Rule-1 violation in the PETSc application ex3.c (Bratu problem)

Factor	First occurrence	Second occurrence
Function Name	FormJacobian	FormJacobian
Line	466	472
Description	if (xs == 0) { /* left boundary */	358
Line	466	472

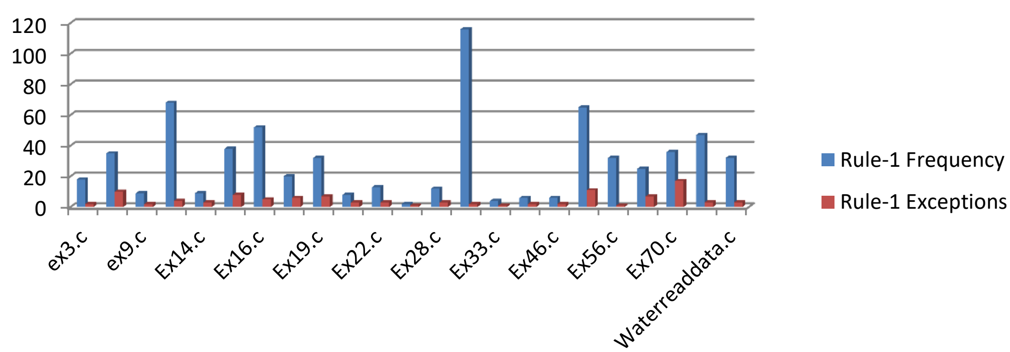

Figure 2: Frequency analysis of Rule-1-related constructs and violations

50% Rule-1 violation rate (i.e., one out of two if-statement constructs triggered Rule-1 violations). As shown in Appendix A, we observe the use of macro constructs only in 11 (out of 46) applications, but we did not find any Rule-2 violations in these applications. Nevertheless, we report these results because they were chosen for this study before we applied our tools and saw the results.

5 THREATS TO VALIDITY

In this study, although we have observed a substantial number of PETSc-based applications, they represent a small portion of the entire PETSc-based code base that exists and is constantly being expanded. Hence, we must explicitly consider threats to the validity of our findings. The first threat is related to generalization of findings. We have reported results after analysis of 46 applications by considering only SNES component of petsc-3.14.3. The inclusion of new core PETSc contributions and new applications in the analysis might alter the presented results. The second threat relates to the scope of the proposed study. We consider only two rules which are related to the use of if-statement and macros constructs of C-usage in PETSc applications. At the start of the study, when our applications sample was even smaller, we did not observe any Rule-2 violations in any application. Hence, inclusion of an analysis of the new rules would affect the interpretation of findings.

6 CONCLUSION AND FUTURE WORK

Identification of developer rule violation can help large open-source HPC projects maintain code quality, which helps with both the long-term sustainability and growth of the software, as well as potentially

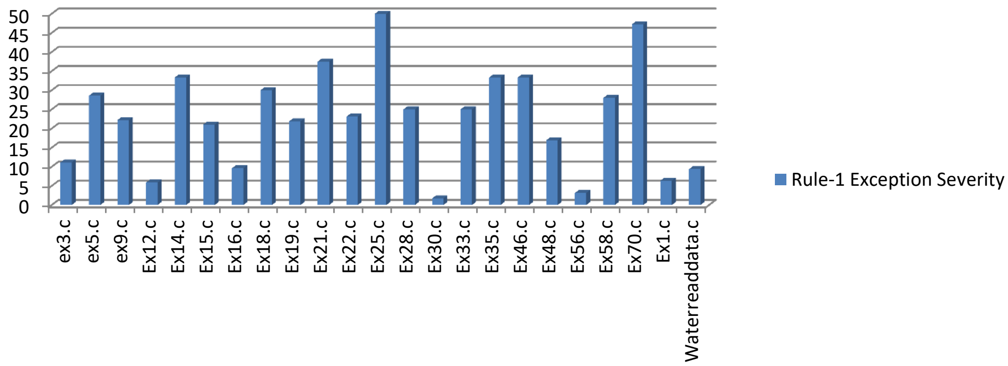

Figure 3: Rule-1 violation severity (percentage of violations vs total occurrences).

improving developer productivity. In this paper, we performed an empirical study for analyzing the developer rule violations in 46 PETSc-based parallel applications that perform simulations of various physical systems described by nonlinear partial differential equations. We included two of the developer C-usage rules defined by the PETSc core developer team and evaluated the violations to these rules in 46 PETSc applications.

The main consequences of our study are as follows.

(1) We identified code matching both rules in most of the applications in our test set.
(2) We observed Rule-1 violations in half of these applications.
(3) Even though we observe use of macros in 11 of the applications, we did not observe any Rule-2 violations.
(4) We observed significantly higher than average numbers of Rule-1 violations in the `ex70.c` and `ex25.c` applications.

In the future, we are planning to expand the set of rules and integrate this analysis into the existing development processes for PETSc and related projects. These checks can be performed for all pull requests, for example, and enable developers to address issues before they become part of a larger code base. Moreover, we will also extend our implementation to include call graph-based analysis to support such rules such as "Array and pointer arguments where the array values are not changed should be labeled as a const argument." Finally, the proposed study could be replicated for PETSc based applications such as ADflow, DEAL.II, MFEM, OpenFOAM, libMesh and MOOSE.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dmitry Karpeyev, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2021. *PETSc Users Manual*. Technical Report ANL-95/11 - Revision 3.15. Argonne National Laboratory. https://www.mcs.anl.gov/petsc
[2] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dmitry Karpeyev, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2021. PETSc Web page. https://www.mcs.anl.gov/petsc. https://www.mcs.anl.gov/petsc
[3] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. 1997. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen (Eds.). Birkhäuser Press, 163–202.
[4] T. Baum, Hendrik Leßmann, and K. Schneider. 2017. The Choice of Code Review Process: A Survey on the State of the Practice. In *PROFES*.
[5] C. Boogerd and L. Moonen. 2008. Assessing the value of coding standards: An empirical study. *2008 IEEE International Conference on Software Maintenance* (2008), 277–286.
[6] Anthony Cox and M. Fisher. 2009. Programming Style: Influences, Factors, and Elements. *2009 Second International Conferences on Advances in Computer-Human Interactions* (2009), 82–89.
[7] S. Eder, Maximilian Junker, B. Hauptmann, E. Jürgens, Rudolf Vaas, and Karl-Heinz Prommer. 2012. How much does unused code matter for maintenance? *2012 34th International Conference on Software Engineering (ICSE)* (2012), 1102–1111.
[8] M. Eichberg, Ben Hermann, M. Mezini, and Leonid Glanz. 2015. Hidden truths in dead software paths. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015).
[9] T. Gee. 2016. Unused Code Detection in IntelliJ IDEA. Available at https://www.youtube.com/watch?v=43-JEsM8QDQ (2019/10/18).
[10] A. Grannan, K. Sood, B. Norris, and A. Dubey. 2020. Understanding the landscape of scientific software used on high-performance computing platforms. *The International Journal of High Performance Computing Applications* 34 (2020), 465 – 477.
[11] Guides. 2020. Importance of Code Quality and Coding Standard in Software Development. Available at https://medium.com/@psengayire.
[12] R. Haas, Rainer Niedermayr, T. Roehm, and S. Apel. 2020. Is Static Analysis Able to Identify Unnecessary Source Code? *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29 (2020), 1 – 23.
[13] G. Holzmann. 2006. The power of 10: Rules for developing safety-critical code. *Computer* 39 (2006), 95–99.
[14] S. Hussain, Jacky W. Keung, and A. Khan. 2017. Software design patterns classification and selection using text categorization approach. *Appl. Soft Comput.* 58 (2017), 225–244.
[15] S. Hussain, Jacky W. Keung, M. Sohail, A. Khan, and M. Ilahi. 2019. Automated framework for classification and selection of software design patterns. *Appl. Soft Comput.* 75 (2019), 1–20.
[16] hypre. [n.d.]. HYPRE: Scalable Linear Solvers and Multigrid Methods. Available at https://computing.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods (1990/01/12).
[17] Yufei Jiang, D. Wu, and Peng Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)* 1 (2016), 12–21.
[18] E. Juergens, Martin Feilkas, Markus Herrmannsdoerfer, F. Deissenboeck, Rudolf Vaas, and Karl-Heinz Prommer. 2011. Feature Profiling for Evolving Systems. *2011 IEEE 19th International Conference on Program Comprehension* (2011), 171–180.
[19] LLVM [n.d.]. The LLVM Compiler Infrastructure. Available at https://llvm.org/.
[20] MISRA-C 2004. MISRA-C 2004 Guidelines for the use of the C language in critical systems. http://www.r-5.org/files/books/computers/languages/c/safe/MISRA-C_Language_Guidelines_for_Critical_Systems_2004-EN.pdf.
[21] PETSc [n.d.]. PETSc Style and Usage Guide. Available at https://docs.petsc.org/en/latest/developers/style/.

[22] Srdan Popic, Gordana Velikic, Hlava Jaroslav, Zvjezdan Spasić, and Marko Vulic. 2018. The Benefits of the Coding Standards Enforcement and it's Influence on the Developers' Coding Behaviour: A Case Study on Two Small Projects. *2018 26th Telecommunications Forum (TELFOR)* (2018), 420–425.

[23] Hashim Sharif, Muhammad Abubakar, A. Gehani, and Fareed Zaffar. 2018. Trimmer: Application Specialization for Code Debloating. *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2018), 329–339.

[24] P. Singayire. 2019. Coding standards and conventions in software development team. Available at https://medium.com/@psengayire.

[25] Ioana Sora. 2015. A PageRank based recommender system for identifying key classes in software systems. *2015 IEEE 10th Jubilee International Symposium on Applied Computational Intelligence and Informatics* (2015), 495–500.

[26] Fabian Streitel, Daniela Steidl, and E. Jürgens. 2014. Dead Code Detection On Class Level. *Softwaretechnik-Trends* 34 (2014).

[27] SuperLU. [n.d.]. SuperLU. Available at https://github.com/xiaoyeli/superlu (1997/04/02).

[28] Trillion. [n.d.]. Trillian - Build Nested CloudStack Environments. Available at https://github.com/shapeblue/Trillian.

[29] Yan-Quing Wang, Z. Y. Qi, Li jie Zhang, and Min-Jing Song. 2011. Research and practice on education of SQA at source code level. *International Journal of Engineering Education* 27 (2011), 70–76.

APPENDIX A

Table 3: Brief descriptions of the test applications used in this study

Application	Directory	Description	Size (Bytes)
ex1.c	tutorials	Use Newton method to solve two-variable systems	11134
ex2.c	tutorials	Use Newton method to solve an equation using Monitor routine	12379
ex3.c	tutorials	Use Newton method to solve an equation using the user-defined Monitoring and linear search routines	12379
ex5.c	tutorials	Use Bratu nonlinear method to solve SFI (Solid Fuel Ignition) problem in 2-D the rectangular domain	38741
ex9.c	tutorials	Solve an obstacle problem in 2D as a variational inequality	11335
ex12.c	tutorials	Solve the Poisson problem in 2D and 3D with simplicial finite elements	105519
ex13.c	tutorials	Solve the Poisson problem in 2D and 3D with finite elements	30849
ex14.c	tutorials	Use Bratu nonlinear method to solve SFI (Solid Fuel Ignition) problem in 3-D using distributed arrays.	20572
ex15.c	tutorials	Use Bratu nonlinear method to solve p-Laplacian	38452
ex16.c	tutorials	An example of large-deformation elasticity buckling	36977
ex17.c	tutorials	Linear elasticity in 2-D and 3-D with finite elements	34506
ex18.c	tutorials	Nonlinear radiative transport with multigrid in 2-D	20373
ex19.c	tutorials	Non-linear driven cavity with multigrid in 2-D	41561
ex21.c	tutorials	Solve PDE optimization problem using full-space method, treats state and adjoint variables separable	7895
ex22.c	tutorials	Solve PDE optimization problem using full-space method, treats state and adjoint variables separable	12192
ex23.c	tutorials	Poisson problem with a split domain	10929
ex24.c	tutorials	Poisson problem is mixed form with 2-D and 3-D with finite elements	20550
ex25.c	tutorials	Minimum surface problem in 2-D	3982
ex28.c	tutorials	1-D multiphysics prototype with analytic Jacobians to solve individual problems and a coupled problem	21954
ex30.c	tutorials	Steady-state 2-D subduction flow, pressure, and temperature solver	59575
ex33.c	tutorials	Multiphase flow in a porous medium in 1-D	6157
ex35.c	tutorials	Laplacian u=b as a non-linear problem	13119
ex42.c	tutorials	Newton's method to solve a two-variable system of the Rosenbrock function	6921
ex46.c	tutorials	Surface process in geophysics	10640
ex48.c	tutorials	Toy hydrostatic ice flow with multigrid in 3-D	74183
ex56.c	tutorials	3-D, tri-quadratic, hexahedra (Q1), displacement finite element formulation	32846
ex58.c	tutorials	Parallel version of the minimum surface real problem in 2-D using DMDA	32846

Continued on next page

Table 3: *Continued from previous page*

Application	Directory	Description	Size (Bytes)
ex59.c	tutorials	Try to solve for an easy case and an impossible case	7295
ex62.c	tutorials	Stroke problem discretized using finite elements	38486
ex63.c	tutorials	Stroke problems in 2-D and 3-D with particles	29976
ex69.c	tutorials	The variable-viscosity strokes problem in 2-D with finite element	182447
ex70.c	tutorials	Poiseuille flow problem, viscous, laminar flow in 2-D channel with parabolic velocity	30550
ex71.c	tutorials	Poiseuille flow in 2-D and 3-D channels with finite elements	21410
ex75.c	tutorials	Variable-viscosity strokes problem in 2-D	20820
ex77.c	tutorials	Non-linear elasticity problem in 3-D with simplicial finite elements	34054
ex78.c	tutorials	Newton methods to solve an equation in parallel with periodic boundary conditions	7794
ex99.c	tutorials	Attempt to solve for roots of a function with multiple local minima	7442
ex1.c	Netowrk	This example demonstrates the use of DMNetwork interface with subnetworks for solving a coupled non-linear	29076
water.c	Water	This example demonstrates the use of DMNetwork interface for solving a steady-state water network model	5403
waterfunction.c	Water	Function used by water.c	7928
waterreaddata.c	Water	Function used by water.c	11544
power.c	power	This example demonstrates the use of DMNetwork interfaces for solving a non-linear electric power grid problem	10871
power2.c	power	This example demonstrates the use of DMNetwork interfaces for solving a non-linear electric power grid problem	26346
pffunctions.c	power	Functions used by power.c	15566
pfreaddata.c	power	Functions used by power.c	7297
ex10.c	ext10d	An example of unstructured grid	29283

Table 4: Rule-1 and Rule-2 findings in application examples included in PETSc.

Application	Directory	# Func.	Rule-1		Rule-2	
			# of if constructs	# violations	# of if constructs	# violations
ex1.c	Tutorials	5	7	0	0	0
ex2.c	Tutorials	5	3	0	0	0
ex3.c	Tutorials	9	18	2	0	0
ex5.c	Tutorials	17	35	10	0	0
ex9.c	Tutorials	7	9	1	1	0
ex12.c	Tutorials	42	68	4	0	0

Continued on next page

Table 4: ***Continued from previous page***

Application	Directory	# Func.	Rule-1		Rule-2	
			# of if constructs	# violations	# of if constructs	# violations
ex13.c	Tutorials	17	14	0	0	0
ex14.c	Tutorials	5	9	3	0	0
ex15.c	Tutorials	11	38	8	0	0
ex16.c	Tutorials	25	52	5	0	0
ex17.c	Tutorials	22	8	0	0	0
ex18.c	Tutorials	4	20	6	0	0
ex19.c	Tutorials	4	32	7	1	0
ex21.c	Tutorials	3	8	3	0	0
ex22.c	Tutorials	7	13	3	0	0
ex23.c	Tutorials	13	5	0	0	0
ex24.c	Tutorials	21	7	0	0	0
ex25.c	Tutorials	2	2	1	0	0
ex28.c	Tutorials	10	12	3	0	0
ex30.c	Tutorials	11	116	2	5	0
ex33.c	Tutorials	3	4	1	1	0
ex35.c	Tutorials	5	6	2	0	0
ex42.c	Tutorials	3	2	0	0	0
ex46.c	Tutorials	6	6	2	0	0
ex48.c	Tutorials	14	65	11	0	0
ex56.c	Tutorials	11	32	1	1	0
ex58.c	Tutorials	7	25	7	0	0
ex59.c	Tutorials	3	3	0	0	0
ex62.c	Tutorials	21	05	0	0	0
ex63.c	Tutorials	22	22	0	0	0
ex69.c	Tutorials	27	20	0	0	0
ex70.c	Tutorials	24	36	17	0	0
ex71.c	Tutorials	15	3	0	0	0
ex75.c	Tutorials	7	3	0	0	0
ex77.c	Tutorials	23	24	0	2	0
ex78.c	Tutorials	3	1	0	0	0
ex99.c	Tutorials	3	3	0	0	0
ex1.c	Network	4	47	3	7	0

Continued on next page

Table 4: *Continued from previous page*

Application	Directory	# Func.	Rule-1		Rule-2	
			# of if constructs	# violations	# of if constructs	# violations
water.c	Water	1	7	0	1	0
waterfunction.c	Water	7	16	0	0	0
waterfunction.c	Water	6	32	3	0	0
power.c	Power	3	12	0	0	0
power2.c	Power	7	38	0	1	0
pffunctions.c	Power	5	33	0	0	0
pfreaddata.c	Power	1	19	0	0	0
ex10.c	ext10d	4	13	0	4	0