Embedded-check: a Code Quality Tool for Automatic Firmware Verification

Rafael Corsi Ferrão*
Insper
São Paulo, SP, Brazil
rafael.corsi@insper.edu

Igor dos Santos Montagner
Insper
São Paulo, SP, Brazil
igorsm1@insper.edu.br

Mariana Silva
University of Illinois
Urbana, Illinois, USA
mfsilva@illinois.edu

Craig Zilles
University of Illinois
Urbana, Illinois, USA
zilles@illinois.edu

Rodolfo Azevedo
Unicamp
Campinas, SP, Brazil
rodolfo.azevedor@ic.unicamp.br

ABSTRACT

Developing embedded microcontroller code is a complex task, especially for undergrad students new to this area. These students often make high-level conceptual mistakes beyond the scope of commercial standards like MISRA-C. These conceptual errors need to be checked manually through code feedback, a process that is time-consuming, error-prone, and does not scale well with an increasing number of students and/or assignments. In this paper, we present an embedded-check an automated tool that can detect common and critical errors students make when learning to code firmware. A set of 13 rules (baremetal and FreeRTOS) was devised based on our experience from several years of teaching Embedded systems. To validate our tool, we compared its results with manual code review of N=99 projects from the last 3 course offerings. We furthered our analysis by running our tool on N=1132 coding lab submissions that did not receive manual feedback and were used as part of classroom activities. We found that the top-3 errors flagged in the projects were already present when students completed the lab activities. We found that (i) our tool also identified all issues discovered during manual code feedback, (ii) our tool detected issues in 86% of student submissions, whereas manual code feedback only flagged 28% of the submissions as problematic, and (iii) 94.3% of students made some code quality error on individual assignments. Within this results, we believe that our tool can have a significant impact when used both as an formative assessment tool to support learning and as a learning analytics tool to improve teaching.

CCS CONCEPTS

• **Software and its engineering**; • **Software creation and management**; • **Software verification and validation**;

KEYWORDS

Embedded Systems, Code Quality Tools, Static Analysis, RTOS

ACM Reference Format:
Rafael Corsi Ferrão, Igor dos Santos Montagner, Mariana Silva, Craig Zilles, and Rodolfo Azevedo. 2024. Embedded-check: a Code Quality Tool for Automatic Firmware Verification. In *Proceedings of the 2024 Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024), July 8–10, 2024, Milan, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649217.3653577

1 INTRODUCTION

Embedded systems play a pivotal role in today's technology-driven world, often operating in critical domains where software reliability is not just a requirement but a necessity[10, 26]. The development of software for embedded systems, therefore, demands specialized attention, as any errors in these systems can lead to catastrophic failures.

To address the critical need for error-free operations in embedded systems, the industry has established a series of stringent standards and norms tailored for specific market niches [5]. Key sectors such as automotive [15], aerospace [30], and medical fields [14] have developed their unique set of standards, which aim to incorporate and enforce good project development practices. These industry-specific standards play a crucial role in guiding the product development process, helping to ensure that safe and correctly functioning products are delivered.

The implementation of these industry standards introduces an additional layer of complexity to the development process. Engineers are tasked with more than just ensuring the functionality of the application; they must also follow specific processes and develop code that adheres to specific rules and guidelines. This requirement for compliance demands a high level of expertise and experience. For instance, the MISRA C [23] standard for embedded software includes 69 pages of required and optional rules. Incorporating it into a course can be overwhelming for students, who are typically encountering embedded systems for the first time.

Designing correct software for embedded systems, especially for novices, poses a significant challenge. Students must master a broad array of topics, ranging from low-level programming languages and cross-compilation to peripheral integration, debugging, and interrupt handling. Furthermore, understanding how these components interact correctly to create an effective solution is crucial.

*Also with Unicamp.

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.
ITiCSE 2024, July 8–10, 2024, Milan, Italy
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0600-4/24/07…$15.00
https://doi.org/10.1145/3649217.3653577

Introducing another topic, such as code and project quality, can be overwhelming in a course already saturated with new concepts.

An alternative to explicitly teaching these code quality's rules is to offer code feedback in students' projects and then require students to refactor their code to fix the issues found by the instructor. This practice, known as code reviewing, is already adopted in both open-source projects [19, 34] and large corporations [29, 32], and it is has been shown to improve software quality metrics [4, 19, 22, 34].

Offering detailed code feedback to each student submission is time-consuming and error-prone, and it is usually offered only a few times per semester, if at all. It also demands a consistency and effectiveness hard to achieve when most of the issues are found by detailed reading of source code and instructors need to check each project twice to see if students fixed the issues correctly.

In this paper, we describe *embedded-check*[1], a tool that automatically checks for common mistakes students make when learning to code firmware. In Section 2, we review related work in embedded systems and education. In Section 3 we describe the course for which the tool was developed. In Section 4, we list 13 rules checked by *embedded-check* and detail which conceptual mistakes they represent. In Section 5, we describe how we validated *embedded-check* and, in Section 6 we present the results. Finally, we present our conclusions, possibilities for use of our tool and work in Section 7.

2 LITERATURE REVIEW

The industry has developed many standards to help ensure quality, safety's and correctness of critical embedded systems, such as DO-178C [30], IEC 61508 [5], ISO 26262 [15]. Regulatory agencies such as the United States' Food and Drug Administration also have published validation guidelines for embedded software [14]. More specifically to the C programming language, MISRA C [23] and BARR-C [3] are the two most used coding standards in embedded software [2]. These standards are large, complex, and represent a body of knowledge aimed at experienced engineers. Even more automated quality models specific to embedded systems such as the one described by Mayr et al. [21] are still too complex for use in classrooms.

Hatton [12] studies the type of restrictions imposed by "safer" subsets of the C language such as the previously cited MISRA C and BARR-C. According to Hatton, many of the rules or guidelines affect languages constructs that *can cause*, but not necessarily lead to, issues. Of these, Hatton differentiates between rules that are based on data and rules that are based on what he calls "folklore" (beliefs that engineers have that are not particularly associated with a known source). Not conforming to these standards do not imply that errors are present, but by adhering to them issues can be avoided. In contrast, static code analysis tools aim to identify parts of the code that have actual problems [36]. Some tools are specific to the challenges present in embedded systems [1, 17, 31] and there has been many successful studies in education as well [7, 8, 24]. Delev and Gjorgjevikj [7] is of special interest since it is based on the same tool as the current work (*cppcheck*), but focuses instead on introductory programming. In this work we focus on issues that arise in embedded systems only and do not address the type of issues that other tools focuses on, such as dead code, unused/unitialized variables, and out-of-bounds errors.

The quality of embedded software has also been a topic of interest for research. Motogna et al. [26] does an empirical assessment of quality on embedded systems, citing that there's so many specific to the area that more traditional approaches are not applicable. When analysing trade-offs between critical characteristics for embedded systems (such as correctness and security [10, 27] and energy efficiency [27, 35]) and more traditional Software Engineering metrics (such as mantainability and reusability), other studies [10, 27, 35] found a preference for the former over the latter.

This same trend seems to be present in embedded systems educational studies. Many previous works [13, 16, 20, 28, 33] describe either pedagogical strategies to increase student learning and engagement and/or the hardware and tooling that supports embedded systems courses. To the best of our knowledge, code quality has not been studied previously in the literature on embedded systems education. This is consistent with the findings in Kirk et al. [18], in which the authors claim the software quality is not present in most of the CS curriculums under study.

3 COURSE CONTEXT

This study was carried out at Insper an small, private university located in Brazil, within a Computer Engineering program. The Embedded Systems course is offered in the fifth semester (of a total of ten) in a sequential curriculum. The course spans 17 weeks and includes two in-person sessions per week, each lasting 2 hours.

The Embedded Systems course focuses on developing firmware in C for microcontroller-based systems. In the first week students take a C workshop taught jointly with other two courses [11, 25]. Core aspects of the language are taught so that, in week two students are already able to start coding simple firmware in C. In this course, we utilize an Atmel/Microchip ARM Cortex-M7 microcontroller (SAME70) along with a development board from the vendor (SAME70-XPLD). The course is taught in C, and we use FreeRTOS as the Real-Time Operating Systems (RTOS). Our primary resources are the vendor's libraries, known as the Advanced Software Framework (ASF), and the vendor's SDK integrated into Visual Studio. We also incorporate additional peripherals according to our needs, which include an OLED display with buttons and LEDs, a color touch LCD, Bluetooth and WiFi modules. The laboratory is well-equipped with devices that are easily accessible to students, who are free to use any of them. The course has its own GitHub repository containing a wealth of example codes (60 examples), which serve as references and templates for students.

The course is structured around two student-centered activities: laboratories and projects. Laboratories are in-class activities where students are introduced to new concepts through solving of concise, well-defined problems. These problems cover various topics such as manual reading, interrupts, Analog-to-Digital Converters (ADC) and RTOS. These activities are formative and help learning the fundamentals of embedded systems. There is typically one laboratory per week, and all are individual assignments.

Projects are designed to challenge students by encouraging them to apply the concepts they have learned in labs to solve more complex and comprehensive problems. These projects require the application of concepts acquired from multiple laboratory experiences to

[1] Avaliable at: https://github.com/rafaelcorsi/embedded-check

develop a unified solution. Additionally, projects provide students with the opportunity to express their creativity and explore new ideas. As a result, most student projects tend to be unique, even when they are based on the same project statement.

The course consists of two projects, which are completed in pairs and typically span one month of development. In the first project, students are tasked with creating a gadget capable of playing monophonic music in a bare-metal environment. For the second project, students are required to design a custom Bluetooth controller for a PC application of their choosing. This project involves the use of a RTOS, and students are also responsible for creating the PC component. Both projects are evaluated using rubrics that encourage students to incorporate more advanced features into their projects.

Starting from the fourth lab, including the second project, the course incorporates the use of a RTOS. From this point onwards, students are required to integrate the RTOS into all their subsequent submissions. Specifically, students are expected to use FreeRTOS to structure their code, employing its features to create tasks for decoupling software components and implementing task-to-task communication, as well as manage interactions between Interrupt Service Routines (ISR) and tasks.

Due to the volume of individual assignments, there is no code feedback for laboratory work. Grades are assigned by a TA who analyzes the self-evaluation and briefly reviews the student's code. In contrast, projects receive detailed code feedback and a thorough examination of the submission. After submission, the instructor reviews all projects and suggests improvements in the code. If a student fails to address the issues or improvements pointed out by the instructor, their grade is capped at a "C", even if they initially achieved a higher mark. If the group makes the corrections and informs the instructor of these changes (by interacting in the issue), their grade is updated to reflect the level achieved in the submission.

4 RULES AND TOOL

Students were introduced to a set of essential rules that they should adhere to in all submissions. These rules, derived from faculty experience and constituting a small subset of what we consider the bare minimum for proficient embedded systems development, were integrated into the content of the labs and the specifications of the projects. They can be categorized into three groups: *C* language, embedded system and FreeRTOS specifics. Table 1 provides a summary of all the coding guidelines along with concise descriptions. In the following sections, we detail each rule and discuss the rationale behind them.

4.1 *C* language

We do not directly teach students how to structure code using *.h* and *.c* files; instead, they need to figure out by themselves how to do this. This set of rules (**noIncludeGuard**, **cInHeadFile**) is aimed at guiding students on how to correctly use these resources. In MISRA C [23], these rules are listed as *E20.1: Header file contents should be protected against multiple inclusions* and *E.8.7: The memory storage (definition) for the variable should not be in a header file*. By enforcing these guidelines, we implicitly encourage students to develop an understanding of modular *C* programming principles. This includes discouraging them from implementing functions in header files and from declaring variables there as well.

4.2 Embedded Systems

In embedded systems, global variables are useful for communicating between interrupt service routines and code running in the central dispatch loop and for holding RTOS resources like queues and semaphores. While we aim to promote *certain* uses of global variables, we also seek to discourage their application as a substitute for passing and returning values between functions through conventional means, as is common in other programming contexts. The rule **wrongUseGlobalVar** is designed to encourage students to properly utilize global variables. In the context of this course, the use of global variables is considered acceptable only when there's a need to allow the communication between an ISR and the main function or to store RTOS resources such as queues and semaphores. This approach aligns with good embedded system practices, where the excessive use of global variables can lead to critical failures [6]. As students are required to use the *volatile* keyword to signal to the compiler that a variable is modified in the interrupt handler, they frequently misuse this keyword on variables that do not require it. To counter this, the **wrongUseofVolatile** rule was implemented. This rule aims to prevent the excessive use of *volatile*, which unnecessarily disables compiler optimizations. Additionally, the **notVolatileVarISR** rule obliges students to correctly apply the *volatile* keyword where it is indeed needed.

Students often attempt to do too much in an interrupt routine, rather than recording what needs to be done and handling it in the central dispatch loop. In a good embedded systems code, the complexity of an ISR shall be kept small and be handled fast; this prevents that another ISR from missing its deadline. This set of rules aims to prevent practices like blinking LEDs (**delayInISR**), updating displays (**oledInISR**), printing text on the terminal (**printfInISR**), or performing pin polling (**whileInISR**). The complexity of an ISR significantly impacts interrupt latency, increasing the risk of missing or delaying other interrupts. Thus, it is imperative to keep ISRs concise and straightforward, steering clear of unnecessary or complex operations such as updating external peripherals or making blocking calls. Most processing work should be shifted to tasks of lower priority or executed in a background loop. These rules are designed to guide students towards more efficient and effective ISR implementations.

4.3 FreeRTOS

FreeRTOS necessitates the use of a specific API when interacting with the RTOS during interruptions (ending with *FromISR*), which can be confusing to students. They must discern when to use each set of functions, a task that may appear straightforward initially but actually demands extensive knowledge of the code. Often, it is not explicitly clear whether a function is part of a software or hardware callback. The rule (**rtosMissingFromISR**) addresses this common oversight in student submissions, which can lead to code failures. It ensures that the appropriate FreeRTOS functions are used within the context of an interrupt service routine, thereby preventing potential issues arising from incorrect API usage.

Rule	Description
noIncludeGuard	All header files shall have include guards.
cInHeadFile	No implementation of code in header files (.h).
wrongUseGlobalVar	Only use global variables to pass information from an ISR to the main function.
notVolatileVarISR	All global variables accessed by an ISR must be declared with the keyword 'volatile'.
wrongUseofVolatile	Only global variables that are modified during the ISR should be global.
delayInISR, *oledInISR*, *printfInISR*, *whileInISR*	Exception handler code (interruption) should be fast, with no blocking/ slow code or external peripheral access.
rtosMissingFromISR	Special functions should be used when calling FreeRTOS functions from an ISR.
rtosWrongUseOfFromISR	When calling FreeRTOS functions from a task, normal functions should be utilized.
rtosWrongUseOfDelay	In most cases, RTOS delays should be preferred over software delays.
rtosWrongUseGlobalVar	When using an RTOS, its resources such as queues and semaphores should be employed.

Table 1: Short description of the code quality rules.

In addition to this, students sometimes mistakenly use these special functions where the normal API should be employed. To ensure correct API usage in such scenarios, the rule **rtosWrongUseOfFromISR** is implemented. This rule verifies that the standard FreeRTOS functions are utilized in the appropriate contexts, preventing the misuse of ISR-specific functions in regular task operations. This dual approach in rule enforcement aids in enhancing students' understanding and correct application of FreeRTOS APIs.

RTOS provides communication resources between distinct parts of the code. Queues and semaphores can be utilized as means of communication between different code segments, such as ISR and tasks/main functions, thereby reducing the need for global variables. This approach not only enhances code security but also improves power efficiency, as it eliminates the need to constantly poll for changes in variables. The use of global variables, other than those providing access to RTOS resources, may indicate poor program structure and a failure to effectively leverage the advantages of an RTOS. The rule **rtosWrongUseGlobalVar** is designed to check for the presence of such variables, ensuring adherence to best practices in RTOS-based program design.

4.4 System Design

The tool is a static code analyzer developed using the abstract syntax tree (AST) generated by the *cppcheck* software [9]. It is implemented as a Python script that utilizes the output generated during code analysis. *Cppcheck* includes a plugin for verifying specific MISRA C rules, and we used this as a foundation for the development of our own rules.

To conduct the analysis, we initially execute *cppcheck –dump* on the *main.c* file of the project. This process generates the AST in the *main.c.dump* file. Subsequently, this file is processed by the *checker.py* script, which is responsible for implementing the rules verification. The script includes an argument *–rtos* that activates the FreeRTOS-specific rules and deactivates the rule **wrongUseGlobalVar** which is replaced by **rtosWrongUseGlobalVar**.

For certain rules, we needed to identify the interrupt callback functions. The embedded C library allows the use of callback functions for various peripherals hardware interruptions (e.g., GPIO, ADC), and these function names are not standardized. To address this, we traversed all the function calls in the code, searching specifically for those that configure these callback functions. Following this, we created a list of these functions, which is accessed when the rule needs to identify, for example, which global variables are accessed from an ISR.

The script includes a configuration file that enables flexibility and customization of certain aspects of the rule. For instance, it allows the use of global variables for specific types of FreeRTOS variables such as queues and semaphores. In this file, we also define which functions are considered critical and should not be used in an ISR. A limitation of the tool is its restricted capability to delve only one level deep in function call analysis. It is not able to verify the implementation of functions beyond this level, such as those called within an ISR. For instance, if a student makes a function call, and within that function, there's another function call or a specific operation like a software delay, the tool would not be capable of performing verification at this second level of function calls.

5 METHODS

We analyzed individual laboratory activities and group project assignments from three course offerings, specifically from the terms Fall 2021 (N=22 students), Spring 2022 (N=48 students), and Fall 2022 (N=26 students) Throughout these courses, we maintained the same core structure, ensuring no significant changes were made to the project or laboratory rubrics. Additionally, all courses were taught by the same faculty member.

For projects, manual code feedback was provided to students through GitHub issues, typically one to two weeks after the submission deadline. This feedback was grounded in the rules outlined on the course website and included any additional aspects the faculty deemed important for student revisions. These encompassed

recommendations for minor code refactoring, the creation of functions, adherence to code style, and corrections of any functional inaccuracies.

These manual feedback on code quality were systematically extracted from the students' repositories and subsequently organized into tables. This data collection provides insights into the extent of effort required by faculty to provide this feedback. We use data from a total of 55 assignments for Project 1 (baremetal), distributed across three semesters, and 49 assignments for Project 2 (RTOS). For each project repository, we extracted several critical pieces of information: the total number of open issues, the number of issues resolved, the types of issues, the categories of errors, and detailed records of student interactions with each issue.

Projects were also analyzed using *embedded-check*, which was developed after students submitted their assignments. For this process, we cloned each student's submission and checked out the git repository to the specific date relevant to their submission before the feedback. The data gathered was utilized to validate the tool by comparing it to the manually generated feedback from faculty. Additionally, the tool was employed on labs submissions to detect and categorize errors that students were making but had not been notified about.

6 RESULTS

The process of manual code feedback resulted in a total of 444 messages from faculty and 263 replies from students spread in 247 issues (65% solved) in Project 1 and 89 issues (44% resolved) in Project 2. Of these, 42 issues (13%) were related to code quality issues that *embedded-check*can check automatically. Table 2 provides a summary of all issue types for each project.

Type	Percentage
Code refactoring	55%
Rules violation	**13%**
Code malfunction	12%
Wrong rubric	7%
Documentation	5%
Solution questions	5%
Complements	3%

Table 2: Percentage Breakdown of Manual Feedback Types of 335 issues.

About half of the manual feedback issues were about code refactoring, which generally relate to better ways to design the code, but not potential correctness issues. The second largest category (13%) was the rules violations that *embedded-check*is design to find automatically. To validate its effectiveness, we ran the tool on student submissions and cross-checked the results with the manual issues opened in students' repositories relating to violations of the specified rules.

In the manual feedback process, typically, one issue is generated for each type of error. For instance, if a student incorrectly uses multiple global variables, the faculty would open only one issue to highlight this mistake. In contrast, the automatic tool generates a warning for each error. To compare the results from both methods, we adopted a binary approach to categorizing errors. If the tool detects, for example, four violations of a particular type, we consider this as a positive indicator of that error in that specific student's repository.

Automatic analysis identified issues 2.6 times more frequently than manual analysis (150 vs 42). In no instance did the manual process detect an error that *embedded-check* failed to identify. Furthermore, in the manual process, we detected some errors in 28% of the repositories, whereas the automatic process, was were able to detect a rule violation in 86% of student submissions. If all the errors detected by the tool had been identified manually, the proportion of issues reported concerning rule violations would have escalated to 34% increasing the significance of these types of errors in student submissions. A detailed comparison between the errors detected in the manual process and those identified by the tool is illustrated in Figure 1.

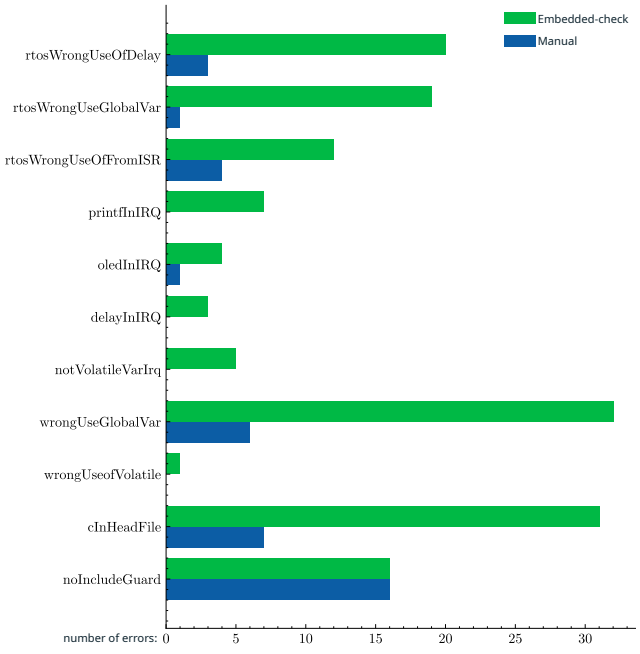

Figure 1: Comparison of Manual x Automatic Rules Violation (just one error of each type is counted per project)

The manual verification process is labor-intensive and relies heavily on the reviewer's attention. The challenge of analyzing a large number of submissions within a limited timeframe significantly impacts the final outcome. The quantity of lines of code further complicates the manual process. In this analysis, student submissions had an average of 353 effective lines of C code, along with 100 blank lines, distributed across 1.8 C files per project. Additionally, there were on average 1264 lines in header files, with 42 blank lines, spread across 3 files per project.

A detailed analysis of the differences between the manual and automatic processes reveals the following:

- **cInHeadFile**: The tool generates a warning for any type of code in header files. Variable declarations and struct initialization often go unnoticed in the manual process.

- **wrongUseGlobalVar**, **notVolatileVarIRS**: The challenge of thoroughly analyzing all code and global variables makes it difficult to individually identify variables that could have reduced scope. Additionally, the challenge of identifying callback functions, which may not have standardized names and could be defined in files different from **main.c**, adds to the difficulty.
- **delayInIRS**, **oledInIRS**, **printfInIRS**: Manual analysis depends on identifying which functions are interrupt and callback functions, complicating error detection.
- **rtosWrongUseOfFromISR**, **rtosWrongUseOfDelay**, **rtosWrongUseGlobalVar**: The discrepancy between individual and automatic analysis could be related to the increased volume of code lines when RTOS is used, which makes identification more challenging.

Analyzing the errors in a non-binarized manner, we detected a total of 383 individualized errors, with the most prevalent being the misuse of global variables, as denoted by the combined occurrences of **rtosWrongUseGlobalVar** and **wrongUseGlobalVar**. Table 3 provides a summary of the results from the automatic analysis, broken down by each specific type of error.

Rule	Projects (%)	Labs (%)
rtosWrongUseOfDelay	15.4	7.0
rtosWrongUseGlobalVar	13.1	55.4
rtosWrongUseOfFromISR	5.0	2.2
rtosMissingFromISR	0.0	0.2
printfInIRS	2.6	6.5
oledInIRS	2.6	1.4
delayInIRS	4.4	4.8
whileInIRS	0.0	0.2
notVolatileVarIRS	1.6	2.2
wrongUseGlobalVar	21.4	9.4
wrongUseVolatile	1.3	4.2
cInHeadFile	19.8	6.2
noIncludeGuard	12.8	0.4

Table 3: Relative frequencies for Rule violations identified in Projects (N = 383 errors) and Labs (N = 1250 errors).

The tool facilitates an analysis of weeakly lab assigments, a task that is not feasible manually due to the large volume of submissions. On average, there were 9.5 submissions per student across a total of 106 students. Theses 1132 submissions (54.4% baremetal and 45.5% RTOS) had a total of 1250 detected errors (see Tab. 3) with a mean of 11.9 violations per student (std=0.7). The tool identified that 94.3% of students made some type of error, with the most common errors being related to RTOS with **rtosWrongUseGlobalVar** (N=692), **wrongUseGlobalVar** (N=118), **rtosWrongUseOfDelay** (N=88), followed by the use of **printf** within ISRs (N=81). Errors that did not appear in the projects, such as **rtosMissingFromISR** (N=2) and **whileInIRS** (N=2), were identified in labs.

In laboratory submissions, there is no requirement to create header files, which leads to fewer errors in this aspect compared to the projects. The errors observed in lab submissions are often due to students creating individual libraries to aid their development process during labs. Additionally, we noticed a correlation between the errors committed in labs and in projects, indicating that students may be carrying over these errors from the conceptual laboratory exercises to the project work.

7 CONCLUSIONS

Code feedback can improve learning but is a time-consuming and error-prone process if done manually by instructors. In this paper we presented *embedded-check*, a tool that checks for common conceptual errors that students make when learning embedded systems. We validated it by comparing its output to manual code feedback given in three previous semesters. Not only *embedded-check*find all problems found by manual inspection, it also detected many more errors, almost tripling the number of detected mistakes. Thus, not only can *embedded-check*save instructors time, but it can also improve the detection rate for mistakes.

The tool can be utilized in various ways to enhance the educational process:

- Post-submission analysis: It can be used for a retrospective analysis of student submissions, helping to identify prevalent errors and assess whether changes in the course curriculum impact the reduction of these errors.
- Assistance in manual code review: The tool can support the manual code review process, making it more efficient and comprehensive.
- Continuous integration in submissions: Implementing the tool as part of a continuous integration (CI) system for individual or group submissions can ensure consistent code quality and adherence to best practices throughout the development process.

embedded-check can be extended to perform new analyses, especially in relation to the use of FreeRTOS by students. It is a frequent occurrence to debug student codes that malfunction due to the utilization of resources such as queues and semaphores without them having been properly created or the use of queues that were created for one data type but are receiving data in a different format. Additionally, examining the issues of *Code Refactoring* type that were manually created can provide insights into new rules that could be implemented in the tool.

When properly justified and thoughtfully considered, certain rules can be made flexible in specific situations. For instance, the use of global variables or the "printf" function within an ISR may not always be impractical; in some cases, it may offer a viable solution to a problem. However, any flexibility from established rules should be approached with caution and careful consideration, aligning with the practices commonly seen in commercial standards.

8 ACKNOWLEDGEMENTS

This work was supported by the Insper-UIUC partnership and the Insper Computer Science Program donors. Their contributions made the 'Adapting Immediate Feedback and Frequent Testing to Project-Based Courses' project possible.

REFERENCES

[1] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. 2018. A runtime verification tool for detecting concurrency bugs in freertos embedded software. In *2018 17th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 172–179.

[2] Roberto Bagnara, Michael Barr, and Patricia M. Hill. 2020. BARR-C:2018 and MISRA C:2012: Synergy Between the Two Most Widely Used C Coding Standards. https://doi.org/10.48550/arXiv.2003.06893 arXiv:2003.06893 [cs]

[3] Michael Barr. 2009. *Embedded C Coding Standard*. Netrino.

[4] Gabriele Bavota and Barbara Russo. 2015. Four eyes are better than two: On the impact of code reviews on software quality. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 81–90.

[5] International Electrotechnical Commission et al. 2010. IEC 61508: 2010-functional safety of electrical/electronical/programmable electronic safety-related systems. (2010).

[6] David M Cummings. 2016. Embedded software under the courtroom microscope: A case study of the Toyota unintended acceleration trial. *IEEE Technology and Society Magazine* 35, 4 (2016), 76–84.

[7] Tomche Delev and Dejan Gjorgjevikj. 2017. Static analysis of source code written by novice programmers. In *2017 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 825–830.

[8] Stephen H Edwards, Nischel Kandru, and Mukund BM Rajagopal. 2017. Investigating static analysis errors in student Java programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. 65–73.

[9] Anum Fatima, Shazia Bibi, and Rida Hanif. 2018. Comparative study on static code analysis tools for c/c++. In *2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*. IEEE, 465–469.

[10] Daniel Feitosa, Apostolos Ampatzoglou, Paris Avgeriou, and Elisa Yumi Nakagawa. 2015. Investigating Quality Trade-offs in Open Source Critical Embedded Systems. In *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures* (New York, NY, USA, 2015-05-04) *(QoSA '15)*. Association for Computing Machinery, 113–122. https://doi.org/10.1145/2737182.2737190

[11] Rafael Corsi Ferrao, Igor Dos Santos Montagner, Ricardo Caceffo, and Rodolfo Azevedo. 2022. How much C can students learn in one week? Experiences teaching C in advanced CS courses. In *2022 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–8.

[12] Les Hatton. 2004. Safer language subsets: an overview and a case history, MISRA C. 46, 7 (2004), 465–472. https://doi.org/10.1016/j.infsof.2003.09.016

[13] Nannan He and Han-Way Huang. 2014. Use of freeRTOS in teaching real-time embedded systems design course. In *2014 ASEE Annual Conference & Exposition*. 24–1307.

[14] Center for Devices {and} Radiological Health. 2020. *General Principles of Software Validation*. https://www.fda.gov/regulatory-information/search-fda-guidance-documents/general-principles-software-validation Publisher: FDA.

[15] ISO. 2018. *ISO 26262-1:2018*. https://www.iso.org/standard/68383.html

[16] JC Jensen, EA Lee, and SA Seshia. 2012. Teaching Embedded Systems the Berkeley Way. In *Workshop on Embedded Systems Education (in conjunction with ESWeek), Tampere, Finland*.

[17] Bjarne Johansson, Alessandro V Papadopoulos, and Thomas Nolte. 2019. Concurrency defect localization in embedded systems using static code analysis: An evaluation. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 7–12.

[18] Diana Kirk, Tyne Crow, Andrew Luxton-Reilly, and Ewan Tempero. 2020. On assuring learning about code quality. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*. 86–94.

[19] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W Godfrey. 2015. Investigating code review quality: Do people and participation matter?. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 111–120.

[20] Akash Kumar, Shakith Fernando, and Rajesh C Panicker. 2013. Project-Based Learning in Embedded Systems Education Using an FPGA Platform. *IEEE Transactions on Education* 56, 4 (2013), 407–415. https://doi.org/10.1109/TE.2013.2246568

[21] Alois Mayr, Reinhold Plösch, Michael Kläs, Constanza Lampasona, and Matthias Saft. 2012. A comprehensive code-based quality model for embedded systems: systematic development and validation by industrial projects. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 281–290.

[22] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21 (2016), 2146–2189.

[23] MISRA. 2023. *Misra C:2023: Guidelines for the use of the C language in critical systems* (3rd, 2nd revision ed.). https://misra.org.uk

[24] Arthur-Jozsef Molnar, Simona Motogna, and Cristina Vlad. 2020. Using static analysis tools to assist student project evaluation. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on Education through Advanced Software Engineering and Artificial Intelligence*. 7–12.

[25] Igor S Montagner, Rafael Corsi Ferrão, Eduardo Marossi, and Fábio J Ayres. 2019. Teaching C programming in context: a joint effort between the Computer Systems, Embedded Computing and Programming Challenges courses. In *2019 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–9.

[26] Simona Motogna, Andreea Vescan, and Camelia Şerban. 2023. Empirical investigation in embedded systems: Quality attributes in general, maintainability in particular. 201 (2023), 111678. https://doi.org/10.1016/j.jss.2023.111678

[27] Lazaros Papadopoulos, Charalampos Marantos, Georgios Digkas, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Dimitrios Soudris. 2018. Interrelations between Software Quality Metrics, Performance and Energy Consumption in Embedded Applications. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems* (New York, NY, USA, 2018-05-28) *(SCOPES '18)*. Association for Computing Machinery, 62–65. https://doi.org/10.1145/3207719.3207736

[28] Sudeep Pasricha. 2022. Embedded systems education in the 2020s: Challenges, reflections, and future directions. In *Proceedings of the Great Lakes Symposium on VLSI 2022*. 519–524.

[29] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice*. 181–190.

[30] RTCA SC-205. 2011. DO-178C - Software Considerations in Airborne Systems and Equipment Certification. https://my.rtca.org/productdetails?id=a1B36000001IcmqEAC

[31] Philipp Dominik Schubert, Paul Gazzillo, Zach Patterson, Julian Braha, Fabian Schiebel, Ben Hermann, Shiyi Wei, and Eric Bodden. 2022. Static data-flow analysis for software product lines in C: Revoking the preprocessor's special role. *Automated Software Engineering* 29, 1 (2022), 35.

[32] Junji Shimagaki, Yasutaka Kamei, Shane McIntosh, Ahmed E Hassan, and Naoyasu Ubayashi. 2016. A study of the quality-impacting practices of modern code review at sony mobile. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 212–221.

[33] André Sanches Fonseca Sobrinho. 2020. An Embedded Systems Remote Course. *Journal of Online Engineering Education* 11, 2 (2020), 01–07.

[34] Christopher Thompson and David Wagner. 2017. A large-scale study of modern code review and security in open source projects. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. 83–92.

[35] Antonio Vetro, Luca Ardito, Giuseppe Procaccianti, Maurizio Morisio, and others. 2013. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. In *Proceedings of ENERGY 2013: The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*. 34–39.

[36] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P Hudepohl, and Mladen A Vouk. 2006. On the value of static analysis for fault detection in software. *IEEE transactions on software engineering* 32, 4 (2006), 240–253.