Static analysis of source code written by novice programmers

Tomche Delev, Dejan Gjorgjevikj
Faculty of Computer Science and Engineering
Ss. Cyril and Methodius University
Skopje
tdelev@finki.ukim.mk, dejan@finki.ukim.mk

Abstract— **In this paper we are reporting the finding on the use of a static analysis of C source code written by students learning to program. Two different tools for static code analysis were used to analyze the solutions submitted by the students on the partial exams and exams from the introductory course in programming in a three year period. We have collected, analyzed and compared most common errors reported by both tools. We further investigate if the available checks provided by these tools, often used in professional software development practices to find bugs and improve the code quality, can also help novice programmers in tracking down and resolving their problems in the code or have any other value in the process of learning programming.**

Keywords— static analysis; novice programmers; programming errors;

I. Introduction

The term static analysis refers to any process of assessing source code without executing it. It is often used by experienced programmers to find bugs, memory leaks or other potential problems in programs. The information reported from static analysis is used to improve code quality, security, robustness, and in some occasions, cyclomatic complexity. Example of errors and problems discovered with static analysis are uninitialized variables, unreachable code, resource leaks, unused variables and many others. Static analysis is also used in dynamically typed programming languages to check for coding standard rules and type errors. In some cases errors discovered with static analysis are indication of serious flaws in program logic and can be potential cause for bugs. For this reason in many complex software projects, the static analysis is automated by including it in the build process. The programmers on the project can use the reports from this analysis to find and fix bugs or improve the code quality. They can also use it to learn about hidden features of the programming language, leading to bugs or unusual behavior of the program [1].

All syllabi for CS majors as well as most syllabi for engineering and science majors do include a course for programming in the introductory year of university studies. Although the freshman now days are much more exposed to computers and technology during high school (secondary education) and some of them have even taken programming courses in high school, programming has not become easier subject. Most researchers agree that in order to master programming one needs some theory and a lot of practice (learning by doing). That is why most programming courses are organized as formal lectures plus a significant number of classes where the students are challenged to solve problems by themselves, usually denoted as laboratory exercises. From the students' point of view this is a form of deliberate practice that is not just simple repetition of the exercise, but challenging oneself with more involving tasks. These tasks should push students beyond their current abilities, and while working on them, they should analyze their performance during and after, and learn and correct the mistakes they are making.

Introductory courses in programming are usually enrolled by large number of students and most of them are with very limited or no programming experience at all. We refer to the students that might have some limited previous experience, or have never before studied or practiced programming as *novice programmers*. Having different background knowledge and aptitude for programming, novice programmers have documented difficulties in learning and understanding programming. These difficulties and misconceptions are often manifested when they try to write programs as part of the formal assessment and examination. Analyzing the source code written by novice programmers can help in discovering the most common errors they make, concepts they find difficult or misconceptions they form.

In this paper, we are exploring the possibility of introducing and applying static analysis on source code written by novice programmers. The goal of this work is to identify the most common error they make and investigate if using tools for static analysis can be valuable to them. We applied two tools for static analysis on dataset of solutions written by novice programmers as part of their exams in a three year period (2013-2015).

The rest of the paper is organized as follows. In section II we present the common challenges of novice programmers when leaning programming and the related research in this area. The used tools for static analysis are described in section III and in section IV, the methodology of the conducted experiment is explained. Finally the experimental results are presented and discussed in section V, and in the last section, a short summary and conclusion is given.

II. Learning Programming Challenges by Novice Programmers

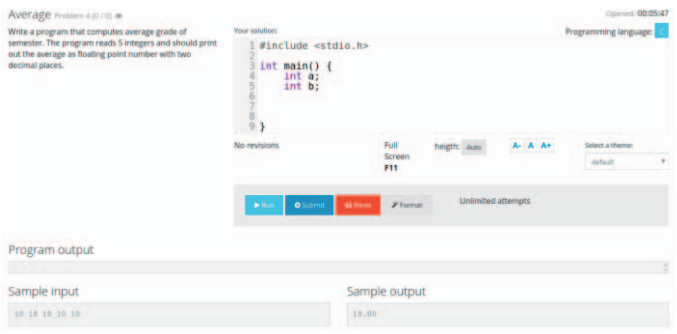
Programming is an individual skill acquired through practice and experience over time. The novice learner, in order to learn to program, must learn the syntax of a programming language and master many skills and concepts, such as constructing a mental model of a notional machine [2]. Novices start writing programs with a very little idea of the properties of the notional machine implied by the constructs of the programming language they are learning [3]. There are many documented cases where novices form faulty mental model of the program dynamics originated from their limited pre-programming knowledge [4]. These limitations can cause difficulties in constructing correct programs and solving problems. For example they can have difficulties in perceiving a piece of code as isolated component rather than an active component of dynamic process that occurs at runtime [5].

To apply static analysis on source code written from novice programmers, first we should introduce the profile of the novice programmer along with their mental model. The novice programmers face many well documented learning challenges in the process of learning programming [6, 7] that we should explore before applying or introducing any new technique in their learning process.

By studying the difficulties, misconceptions or common errors the novice programmers are making we can achieve better understanding of the problem-solving strategies and highlight the difficult aspects of programming. Furthermore, such studies can spark contributions in refining of existing or invention of new programming languages, training tools and teaching methods. One approach on identifying the misconceptions of programming is by interviewing students using think-aloud protocol on closed list of problems covering different concepts such as control flow, types, conditionals and others [8]. Other approach can be by polling the students with questionnaire on different topics and concepts. For example one such study [7] shows that topics that rely on clear understanding of pointers and memory-related concepts proved to be the most difficult. The approach we have taken in this paper was to collect and process the reports of applying static analysis on the code written by novices, and then quantify and analyze the reported errors. The analyzed source code files are solutions submitted by novice programmers on problems given on partial exams and exams.

For many novice programmers, the first interaction with the tools needed to compile, run and test their program is challenging. To address this issue, in the past years in the field of teaching and learning programming many automated programming assessment systems are gaining popularity [9]. Such systems are considered to significantly lower the entry barrier for novice programmers caused by complicated tools or integrated development environments (IDE's). Our solution to these problems, is a system named Code [10], developed at the Faculty of Computer Science and Engineering at the Ss. Cyril and Methodius University in Skopje, used for automatic assessment and management of students' programming exercises and exams. The system provides a simple web-based user interface, where the students can write, run and test their solutions (fig. 1). The solutions are automatically stored, compiled and executed on a central server. One of the main advantages of the system is the automatic assessment with immediate feedback, which can help both, the course instructors and the students. Providing timely and informative feedback in such systems is important component [11] especially in situations when direct feedback from instructors is not feasible. The feedback can vary from plain compilation output, to hints of errors in the program, or listing and comparing the test input data and the expected output. The type of feedback provided is important and can affect the strategy imposed in the process of constructing a working solution.

One idea investigated in this work is the possibility of using static analysis tools to enrich the feedback with valuable information for potential bugs and errors. Static analysis can be used to check programming style, program errors (syntax or semantic), software metrics assessment, structural and non-structural similarity analysis, keywords analysis, plagiarism detection and diagram assessment [12]. It has also been successfully used to help students writing better code [13].


Average Problem 4 (0 / 0)

Opened: 00:05:47

Write a program that computes average grade of semester. The program reads 5 integers and should print out the average as floating point number with two decimal places.

Your solution:

Programming language: C

```
#include <stdio.h>

int main() {
    int a;
    int b;



}
```

No revisions

Program output

Sample input

```
10 10 10 10 10
```

Sample output

```
10.00
```

Fig. 1. Web-based interface of the automatic assessment system Code

III. Static Analysis Tools

Static analysis is performed with specialized tools developed only for that purpose. A static analysis tool can explore large number of “what if” scenarios without having to go through all the computations necessary to execute the code for all scenarios. Also, good static analysis tools provide a fast way to create complete and consistent evaluation report of the source code. Most often these tools are specialized for evaluation of source files written in single programming language. However, there are some more general tools capable of evaluating source files in multiple, often similar programming languages.

The source code of the solutions analyzed in this study was written in the C programming language. We have tried the following tools for static analysis of C code: Clang Static Analyzer, CppCheck, Splint, OClint and the first two were chosen for this analysis. Our motivation was to select the most appropriate ones for our context of introduction level of programming and the programming language C, and not to review tools for static analysis of C source code. The available checks and the reporting of errors these tools can provide, also was an important factor in this decision. Clang Static Analyzer and CppCheck were chosen as they were found to provide distinct types of checks and warnings about potential source code problems. The first one also includes some recently introduced experimental checkers relevant to our research, and also it's used as underlying checker of one of the remaining static analysis tools OClint.

A. Clang Static Analyzer

The Clang Static Analyzer [14] is a source code analysis tool part of Clang which is a C, C++ and Objective-C front-end of the LLVM compiler [15]. The Clang Static Analyzer can spot errors and bugs with different degrees of analysis sophistication, mostly using the control-flow graph of the program. Finding errors and potential bugs is implemented by various checkers that are grouped in the following six groups: *core checkers*, *C++ checkers*, *dead code checkers*, *OS X checkers*, *security checkers* and *UNIX checkers*.

In our analysis we used the core checkers which model core language features and perform general-purpose checks such as division by zero, null pointer dereference, usage of uninitialized values and errors. We also used dead code checkers and some of the experimental checkers. Example of some of these checks are shown in table 1.

B. CppCheck

CppCheck is a static analysis tool for C/C++ code which primarily detects the types of bugs that the compilers normally do not detect, such as: *out of bounds checking*, *memory leaks checking*, *detect possible null pointer dereferences*, *check for uninitialized variables*, and few others. CppCheck can also be extended with simple patterns, defining rules for functions or scripts.

TABLE I. EXAMPLE CHECKS FROM CLANG STATIC ANALYZER

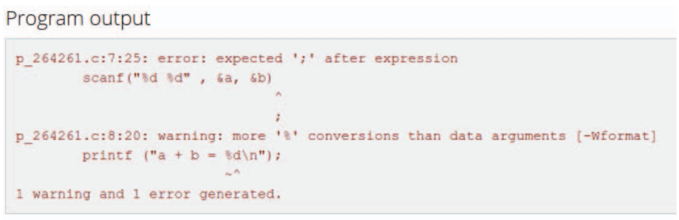
Name, Description	**Example**
core.CallAndMessage (C, C++, ObjC) Check for logical errors for function calls (e.g., uninitialized arguments, null function pointers).	`void f(int x);` `void test() {` `int x;` `f(x); // warn: passed-by-value // arg contain uninitialized data` `}`
alpha.deadcode.UnreachableCode (C, C++, ObjC) Check unreachable code.	`int test() {` `int x = 1;` `while(x);` `return x; // warn` `}`

IV. METHODOLOGY

The study is based on the data collected during an introductory programming course "Structured programming" using the C programming language. The enrollment (including re-enrollment) on this course reaches up to 1000 students. The course covers the basic programming constructs and structured programming concepts in 14 weeks with 2 forty-five minutes long lectures held by professors and 2 equally long problems solving sessions given by teaching assistants. The concepts covered are reading and writing to I/O, flow control, iteration and recursion, arrays (two-dimensional), pointers and reading or writing files. Most of the example problems can be solved by constructing a short and simple straightforward algorithm.

Students also participate in 10 (hour and a half long) hands-on laboratory sessions where they are presented with several exercise problems that they supposed to solve individually. The assessment is organized in two partial exams and a final exam (during the semester), followed by two additional final exams sessions during the academic year. The exams usually include up to four problems, on which the students are expected to write complete working solutions. Exams are taken in controlled laboratory environment, where students can write, run and test their solutions in the web-based environment for programming and automatic assessment Code. When they open a problem in the system Code, they have a side by side view of the problem text and a web-based text editor (fig. 1), where they can write their solution. Students also can use other editors or IDE's to compile and execute their solutions on the local machine, but are required to post the final solution in the system. Once a student has written a solution, he/she can make one of two possible actions, *run* or *submit*. On the action *run*, the solution is first compiled and then if the compilation is successful, it's executed with a sample test input which is visible in the problem view. In case of compilation errors the output of the compiler is presented, and otherwise the output of the execution is presented next to the expected output for the given test input. The students can visually compare both, the output from the execution of their program and the given expected output. On the *submit* action, the solution is compiled and then executed with multiple test cases as input. For each test case the output of the execution of the students' solution is compared to the expected output for that test case, and a report of the comparison is presented to the student. Some of the test cases can be hidden (the input, the expected and actual output are not shown) and only the result (passed/failed) is reported. Students can make unlimited number of submit attempts.

On each run or submit attempt the system automatically provides three types of feedback. The first type of feedback is the output from the compilation process before execution with the standard report from the GCC or Clang compiler (fig. 2).

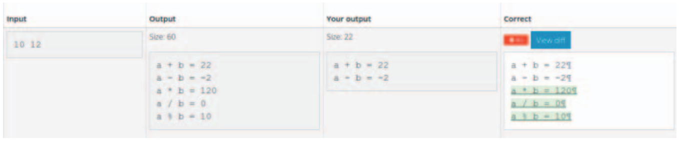


Program output

```
p_264261.c:7:25: error: expected ';' after expression
        scanf("%d %d" , &a, &b)
                               ^
                               ;
p_264261.c:8:20: warning: more '%' conversions than data arguments [-Wformat]
        printf ("a + b = %d\n");
                         ~^
1 warning and 1 error generated.
```

Fig. 2. Compilation output feedback

The second type of feedback is the report from dynamic analysis of their solutions (fig. 3). The dynamic analysis is testing the correctness of the solution by comparing the output from the execution with a predefined test cases for the problem. Also, as third type of feedback an HTML report from analyzing the source code with the Clang Static Analyzer is available.



Input	Output	Your output	Correct
10 12	Size: 60	Size: 22	View diff
	a + b = 22	a + b = 22	a + b = 22¶
	a - b = -2	a - b = -2	a - b = -2¶
	a * b = 120		a * b = 120¶
	a / b = 0		a / b = 0¶
	a % b = 10		a % b = 10¶

Fig. 3. Dynamic analysis feedback for solution.

In this study we have assembled a dataset containing students' solutions on the problems given on the exams in the academic years 2013, 2014 and 2015. To build the dataset, a total of 13,960 source code files submitted as final solutions to the exam problems were extracted from the system Code. We used the selected tools for static analysis on the full dataset, and analyzed the generated reports. The Clang Static Analyzer can only be performed on source code files that can be compiled successfully, while all source code files (including those that cannot be compiled) can be processed with CppCheck. Both tools can be used as command-line programs that can be executed on a group of files and they report the results in semi-structured format. The Clang Static Analyzer also outputs a visual report in HTML file, where the found errors are described in more details. The data used for reports was extracted by processing and parsing the reports generated by using the static analysis tools on the dataset.

From our observations on a sample of the results from the exams, we can identify four groups of students by their performance. The first and usually small group of talented students, with almost perfect score, are the ones who solve the exam problems without any obstacles. The second group, students who passed the exam with solving at least one problem without mistakes, consist of students who can solve most of the problems with some obstacles, but might make small mistakes on the path. The third group of students, is identified as the group that can write some working code for simple assignments, but have problems to create or deduce a clear mental model for more complicated programming constructs such as arrays, nested loops or recursion. The fourth and last group, cannot write any working code and in most cases is unmotivated to perform.

V. Experimental Results

A. Compilation results

As a first step of the analysis, compilation was attempted on each solution of the dataset. Here we report the results of compilation success rate for each exam. Only successfully compiled solutions were used in the further analysis with the Clang Static Analyzer. The results showed that almost 25% of the source files contained at least one compilation error. Significant percentage of them were empty or contained some free text (detailed results are shown in table 2). Errors in compilation are indication of lack of knowledge of the syntax of the C programming language. Also some students have manifested difficulties in understanding and then acting upon the compiler messages. There are also few cases when students are not motivated or interested in writing solution of the problem, so they write random content that does not compile.

TABLE II. PERCENTAGE OF SUCCESSFULLY COMPILED SOLUTIONS

Year	Partial exam 1	Partial exam 2	January exam	June exam	August exam
2013	74.1%	81.7%	73.4%	69.6%	73.3%
2014	74.8%	81.6%	71.8%	68.3%	73.3%
2015	79.1%	87.0%	64.8%	64.5%	75.8%

On figure 3 the overall compilation success over the time of one year is shown, starting from the first partial exam in November and ending with the last exam in August. The highest rate in compilation on the second partial exam, can be explained by the fact that only students that scored over 40% on the first partial exam are allowed to take the second. These are usually students that have already proven their basic knowledge in programming and the programming language syntax. The final exam is only taken by the students that have scored less than the passing minimum of the partial exams. Also, noticeable is the consistency in the change of the compilation success rate on same exams across the analyzed period of three years.

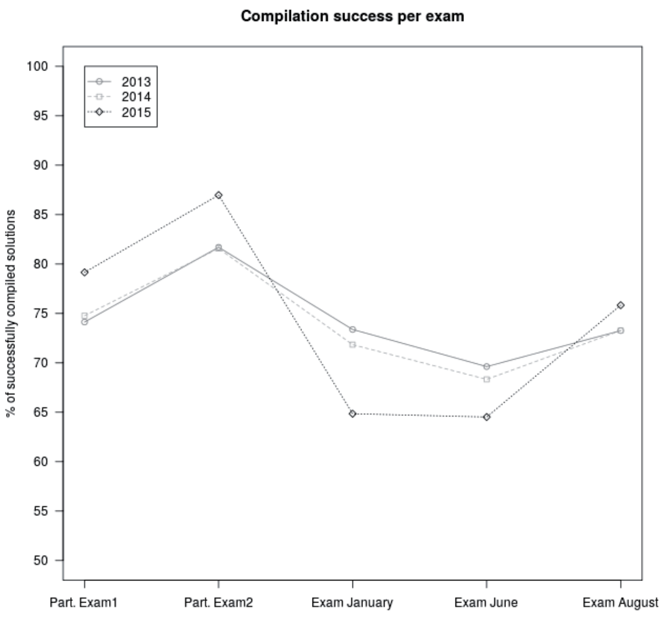
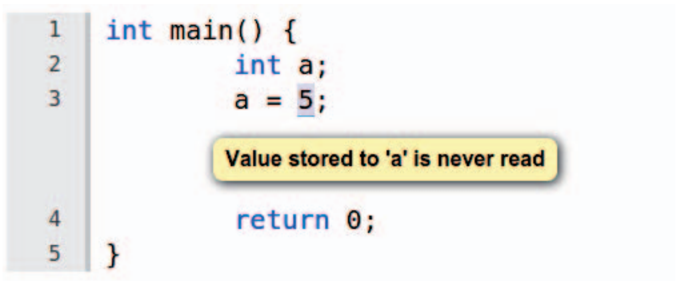


Fig. 4. Compilation success over academic year

B. Clang Static Analysis report results

Running Clang Static Analysis tool on the dataset, produced detailed report with all the detected errors for each source file containing errors. The errors are grouped by two types of checks, denoted as *Dead store* and *Logic errors*.

Dead store check includes type of checks such as dead assignment, dead initializations or dead increment. Dead assignment (fig. 5) is a common error where a variable is assigned some value, and later in the code the assigned variable is never used. Dead initialization is a similar error where variable is initialized with some value, and in the rest of the code that variable is never used. Dead increment is a type of error where a variable is mutated (incremented or decremented), and then this new value of the variable is never used. The distribution of these errors it's shown on figure 6. These kind of errors such as incorrect variable declaration and usage are very common between novice programmers and in many cases are serious indication of a wrong solution. Early reporting on such errors can be helpful in finding a potential bug caused by this kind of errors. From all the errors reported, 34% turned out to be dead store errors.



```
int main() {
        int a;
        a = 5;
            Value stored to 'a' is never read
        return 0;
}
```

Fig. 5. Dead assignment error

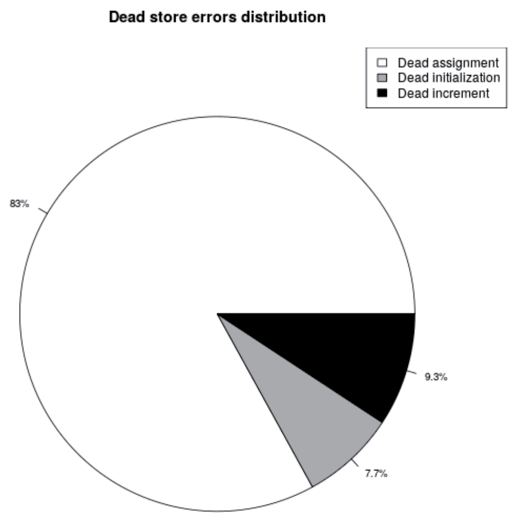
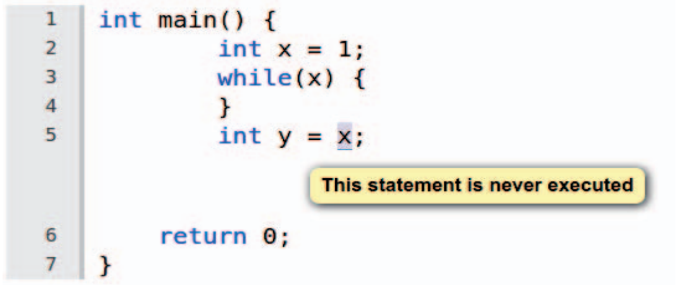


Fig. 6. Dead store errors distribution

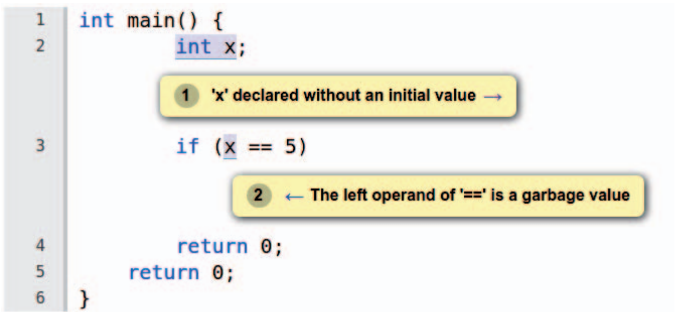
Dead code is experimental check for unreachable code, which is a block of code that will never be executed, often as a consequence of previous endless loop or incorrect branching condition. This type of error is difficult to find and determine in compilation time using only static analysis. For example using the function *scanf* for reading from standard input can be a source for false positives or missed endless loops. This error (when found successfully) is a clear indication of wrong solution, or wrong implementation of an algorithm. This can be very useful information for novice programmers, because it is type of error not commonly found by compilers. To be able to find these kind of errors, which are usually found at runtime, requires very good understanding of the program dynamics and execution flow. Example of reported dead code error is shown on figure 7.



```
int main() {
        int x = 1;
        while(x) {
        }
        int y = x;
            This statement is never executed
    return 0;
}
```

Fig. 7. Dead code error

Logic errors are the most common type of errors made by novice programmers. As shown on table 3, 66% of all errors are logic errors. Of all logic errors detected, following four types are the most common: *result of operation is garbage or undefined*, *uninitialized argument value*, *assigned value is garbage or undefined* and *out-of-bound access*. The precise distribution can be seen on figure 9, where the fifth group denoted as *"other"* includes all other reported errors. Errors such as these are related with concepts such as conditional logic, arithmetic operations, and variable initializations and updating. The Clang Static Analyzer does a very good job in tracking the logic errors through the code, and traces the steps that have caused the error (as shown on figure 8). The report produced by this analyzer can be especially helpful to novices not just for locating the error, but also for learning and understanding how it happened in the first place.



```
int main() {
        int x;
            1  'x' declared without an initial value →
        if (x == 5)
                2  ← The left operand of '==' is a garbage value
        return 0;
    return 0;
}
```

Fig. 8. Logic errors

The total number of errors and percentages of source files with at least one error found with static analysis using the Clang Static Analyzer are presented on table 3. Significant percentage of the *dead store* error were false positives, caused by the above mentioned problem with the *scanf* function. Still, the large numbers are showing that using this tool can be helpful for students to find many types of errors, they often fail to notice with standard compiler or just by running and inspecting the code.

TABLE III. CLANG STATIC ANALYZER ERRORS REPORT

Year	Dead store errors	Logic errors	Total solutions analyzed	% of solutions with at least one error
2013	784	2,369	3,333	54.06%
2014	827	2,646	3,552	57.48%
2015	1,001	2,675	3,477	61.75%
Total	2,612 (34%)	7,690 (66%)	10,362	57.56%

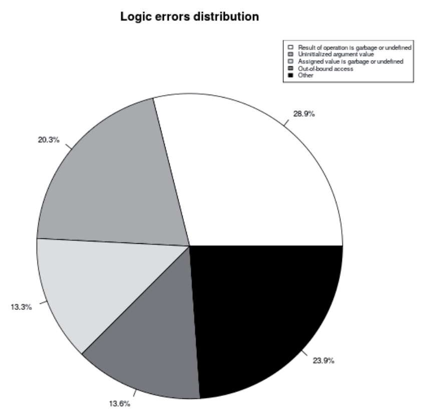


Fig. 9. Logic errors distribution

C. *CppCheck results*

CppCheck static analysis tool produces reports on 5 groups of errors: *performance*, *style*, *warning*, *portability* and *error*. In our study we ignored the performance and portability checks as not relevant for the novice programmers and focused only on the other three types. In table 4 the most common types of checks reported by CppCheck are summarized.

TABLE IV. TYPES OF CHECKS FROM CPPCHECK

Type	Example
Style	Unused variables The scope of variable ‘x’ can be reduced Variable ‘x’ is assigned a value that is never used Variable ‘x’ is not assigned a value
Warning	%d in format string (no. 1) requires 'int *' but the argument type is 'int' printf format string requires 0 parameters but 1 is given %d in format string (no. 1) requires 'int' but the argument type is 'int *' String literal compared with variable 'n'. Did you intend to use strcmp() instead? Comparison of a boolean expression with an integer other than 0 or 1
Error	Uninitialized variable Invalid number of character ({) when these macros are defined Array index -1 is out of bounds

The errors reported from this tool includes unused variables, wrong comparison expressions, array indexes out of bounds and others that are common between novice programmers. The results on table 5 are showing that novice programmers are making large number of these errors, with an average of more than 2 (style/warning/error) per solution.

TABLE V. RESULTS FROM STATIC ANALYSIS USING CPPCHECK

Year	Total solutions	Style	Warning	Error
2013	4,491	5,512	1,772	2,183
2014	4,815	6,435	1,139	2,609
2015	4,654	6,381	1,394	2,645

VI. DISCUSSION AND FUTURE WORK

Debugging and finding errors is proven to be very difficult task, which is particularly difficult and frustrating for novice programmers. Students have indicated that the easiest bugs to fix are those found by the compiler or some other tool. This suggests that for students, locating bugs is more difficult than fixing them, or that the types of bugs found by the compiler and other tools, which are most often construct-related, are easier to fix than other bugs [16]. The results reported in this paper, are confirming these findings, by showing that tools for static analysis can be helpful to novice students in finding and understanding bugs. Incorporating such tool in the students' learning environment for programming such as Code, can bring multiple advantages. It can help students to find logic errors, learn about them and consequently make them better in the debugging process.

Static analysis is proven and widely used tool by professionals to find bugs and code smells in large code bases. It is mostly used to help avoiding bed practices or find and track down bugs. In this study we have used two industry tools for static analysis to process the source code written from novice programmers during exam sessions. The results are confirming that novice programmers make large number of mistakes such as “uninitialized variable”, or styling types of error such as “unused variable”. Most of these errors are left unnoticed by novices, and are often cause for wrong solutions, even when their initial idea for the algorithm is correct. This study shows that including static analysis in the learning process can have value, mostly because it can help novices in finding individually many common errors they make. How will they actually accept and act on the reports on the errors can be an interesting question we can try to answer in some future work.

ACKNOWLEDGMENT

The work presented in this paper was partially financed by the Faculty of Computer Science and Engineering at the Ss. Cyril and Methodius University in Skopje, Macedonia.

REFERENCES

[1] Emanuelsson, P., Nilsson, U.: A comparative study of industrial static analysis tools. Electronic notes in theoretical computer science 217, 5–21 (2008).

[2] Sorva, J.: Notional machines and introductory programming education. ACM Transactions on Computing Education (TOCE) 13(2), 8 (2013)

[3] Du Boulay, B., O'Shea, T., Monk, J.: The black box inside the glass box: presenting computing concepts to novices. International Journal of Man-Machine Studies 14(3), 237-249 (1981)

[4] Bonar, J., Soloway, E.: Preprogramming knowledge: A major source of misconceptions in novice programmers. Human-Computer Interaction 1(2), 133-161 (1985)

[5] Eckerdal, A., Thun, M.: Novice java programmers' conceptions of object and class, and variation theory. In: ACM SIGCSE Bulletin. vol. 37, pp. 89-93. ACM (2005)

[6] Lahtinen, E., Ala-Mutka, K., Jarvinen, H.M.: A study of the dificulties of novice programmers. In: ACM SIGCSE Bulletin. vol. 37, pp. 14-18. ACM (2005)

[7] Milne, I., Rowe, G.: Difculties in learning and teaching programming views of students and tutors. Education and Information technologies 7(1), 55-66 (2002)

[8] Kaczmarczyk, L.C., Petrick, E.R., East, J.P., Herman, G.L.: Identifying student misconceptions of programming. In: Proceedings of the 41st ACM technical symposium on Computer science education. pp. 107-111. ACM (2010)

[9] Ala-Mutka, K.M.: A survey of automated assessment approaches for programming assignments. Computer science education 15(2), 83{102 (2005)

[10] Delev, T., Gjorgjevikj, D.: E-lab: Web based system for automatic assessment of programming problems. Web proceedings ICT-Innovations (2012).

[11] Venables, A., Haywood, L.: Programming students need instant feedback! In: Proceedings of the 5fth Australasian conference on Computing education-Volume 20. pp. 267-272. Australian Computer Society, Inc. (2003)

[12] Rahman, K.A., Nordin, M.J.: A review on the static analysis approach in the automated programming assessment systems. In: Proceedings of the national conference on programming. vol. 7 (2007)

[13] Truong, N., Roe, P., Bancroft, P.: Static analysis of students' java programs. In: Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30. pp. 317–325. Australian Computer Society, Inc. (2004).

[14] Kremenek, Ted. "Finding software bugs with the clang static analyzer." California: Apple Inc (2008).

[15] Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Code Generation and Optimization, 2004. CGO 2004. International Symposium on. pp. 75–86. IEEE (2004)

[16] Fitzgerald, Sue, et al. "Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers." Computer Science Education 18.2 (2008): 93-11