STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVIII**, Number 1, 2023
DOI: 10.24193/subbi.2023.1.03

DETECTING PROGRAMMING FLAWS IN STUDENT SUBMISSIONS WITH STATIC SOURCE CODE ANALYSIS

PÉTER KASZAB AND MÁTÉ CSERÉP

Abstract. Static code analyzer tools can detect several programming mistakes, that would lead to run-time errors. Such tools can also detect violations of the conventions and guidelines of the given programming language. Thus, the feedback provided by these tools can be valuable for both students and instructors in computer science education. In our paper, we evaluated over 5000 student submissions from the last two years written in C++ and C# programming languages at Eötvös Loránd University Faculty of Informatics (Budapest, Hungary), by executing various static code analyzers on them. From the findings of the analyzers, we highlight some of the most typical and serious issues. Based on these results, we argue to include static analysis of programming submissions in automated and assisted semi-automatic evaluating and grading systems at universities, as these could increase the quality of programming assignments and raise the attention of students on various otherwise missed bugs and other programming errors.

1. Introduction

The demand for IT professionals is constantly increasing, as software development and maintenance is required in various fields, ranging from the finance sector through energy and manufacturing to healthcare [9]. As a direct consequence, more and more people are enrolling each year in computer science degree programs and other IT and programming related courses at universities [14]. This increment of students significantly increases the workload of university teachers and makes the manual grading of each student submission

Received by the editors: 01 March 2023.

2010 *Mathematics Subject Classification.* 68U99, 68Q55, 97Q70.

1998 *CR Categories and Descriptors.* F.3.2 [**Theory of Computation**]: Logics and Meanings of Programs – *Semantics of Programming Languages*; D.3.4 [**Software**]: Programming Languages – *Processors*; K.3.2 [**Computing Milieux**]: Computers and Education – *Computer and Information Science Education.*

Key words and phrases. static code analysis, C++, C#, student submission, computer science education, programming flaw.

unsustainable. As a result, the usage of automatic grading systems for programming assignments have gained focus in the past years. Whether they are developed commercially, open-source or in many cases as an internal project at a university, these systems are becoming indispensable for instructors [11].

Non-trivial run-time errors in programming submissions are often missed by instructors and automatic testers, because these kinds of errors are not always easy to find and reproduce. Furthermore, there are solutions with functionally correct and bug-free code which do not follow the conventions and guidelines of the given programming language. These kinds of errors can be avoided and the application of the given guidelines can be forced using static code analyzers [4, 12, 18, 22].

Static analyzer tools can be utilized to check or flaws in other programs. This can be achieved by evaluating the source code, the byte code or the binaries [2]. These tools can help developers to identify a wide range of issues from incorrect styling and formatting to serious security issues [10]. In Section 2 we review the previous applications of such tools in higher education.

For our research, we evaluated over 5000 student submissions from the last two years written in C++ and C# programming languages at the Eötvös Loránd University Faculty of Informatics (*ELTE FI*), by executing various static code analyzers on them. In Section 3 we introduce the evaluated courses, the used analyzer tools and the criteria for filtering analyzer results. Then, in Section 4 and 5 the most typical and interesting findings are presented for the C++ and C# submissions with simple code examples. We showcase our prototype implementation for an automated evaluator system in Section 6. Finally, the conclusion and future work is described in Section 7.

2. Related work

2.1. **Automatic evaluation of submissions.** Static analyzers can be used in education in order to help the learning process of the students and speed up the evaluation of the submission.

Michael Striewe and Michael Goedicke [22] reviewed the static analysis approaches that can help in providing feedback to the submitted solutions. They highlighted the following requirements for a system that evaluates submissions with static analysis:

- check for mistakes and violated coding conventions in syntactically correct source code;
- check for source code which is correct, but contains element that are not allowed in the context of the given course or exercise;
- check for missing code structures;
- give hints on how to solve the previously mentioned issues.

J. Walker Orr [19] proposes a rule-based tool for Java and Python that provides feedback on predefined rules. The checked design principles are formalized as logical functions, and they are applied to the subtrees of the abstract syntax trees. The implemented rules are designed to meet the needs of students. The system was hosted as a standalone web service where students could submit their solutions. There were no limitations to the number of uploads, and the execution of the tests was instant. Thus, this increased the transparency of the grading progress. On average, the rate of design quality flaws dropped 47.84% on different assignments.

Blau et al. developed a tool called *FrenchPress* [4] which is an Eclipse-plugin designed for students with intermediate knowledge of the Java programming language. It focuses on silent flaws that often get overlooked by students, because IDEs and compilers do not catch them. The advantage of the IDE integration, that the students can get feedback while they work in their code without leaving the development environment. The authors emphasize that the feedbacks should be relevant for the situation of the students and should be easy to understand. Also, it is important to minimize the number of false positives, as they could be more problematic than false negatives for inexperienced users. In the end, the percentage of cases when *FrencPress* motivated the users to improve their programs varied from 36% to 64% depending on the course.

In contrast to the previously mentioned tools, *Hyperstyle* [3] uses existing professional code analyzers to evaluate the submissions. It currently supports Java, Kotlin, JavaScript and Python, but it can be extended easily with analyzers for other languages. The possible errors are split into the following categories: code style, code complexity, error-proneness, best practices, and minor issues. Based on the findings, it gives grades for the solutions on a four level scale: excellent, good, moderate, bad. Additionally, it provides custom messages for some issues, because students often need more detailed errors messages than the output of the professional tool. *Hyperstyle* was tested on the *Stepik* and *Jetbrains Academy* platforms, but it can be added to other MOOC systems as well. For Java solutions, the number of students who made fewer mistakes increased, and the number of who made six or more mistakes decreased. For Python solutions, the number of students without code quality issues increased four times and the number of students who made two or more errors decreased.

2.2. **Analyze solutions from previous semesters.** While the previously reviewed papers present solutions that provide feedback to students or grade their submissions automatically, analyzing datasets of existing submission can provide valuable information on several aspects. Moreover, checking older solutions can also help to evaluate the code analyzer tools, and adapt their results to the needs of the students and teachers.

Molnar et al. [18] evaluated Python assignments from an introductory programming course using *Pylint*. They also developed a custom tooling that is able to visualize and list findings for: a specific student, a given assignment, or an assignment corresponding to multiple students. Their study showed that *Pylint* provided meaningful information regarding code style and logical errors.

Keuning et al. [12] investigated code quality issues in Java programs. The analyzed source files were collected from the *Blackbox* database, which contains Java solutions written in the *BlueJ IDE*. The database stores multiple versions of the source files, and also collects events and usage statistics from the *BlueJ IDE* (e.g., enabled plugins). They used the *PMD* tool for analysis and categorized the errors into the following categories: flow, idiom, expression, decomposition, and modularization. Some detectable issues by *PMD* needed to be discarded, because they were too advanced for novice programmers, or they were too specific for a library or platform. Errors from categories like presentation and documentation were completely discarded. They also examined the most commonly fixed errors. Empty if statements and singular fields were the most commonly fixed issues. Probably, because the initial uploads were not finished. Issues like too many fields or methods were fixed in less than 5% of the cases. So, the overall fixing rate was relatively low.

Similarly to the previous paper, Edwards et al. [6] also analyzed Java programs from four different courses for students with different skill-levels. The dataset contained nearly 10 million errors produced by 3691 students. They used the *CheckStyle* and *PMD* open-source tools for static code analysis, but they created their own categories for errors: braces, coding flaws, documentation, excessive coding, formatting, naming, readability, style, and testing. The most common categories were documentation, formatting, style, and coding flaws. The coding flaws category could indicate that the student is struggling. Usually, the solutions with lower scores had more coding flaws in them. Also, it is possible that some students ignore the warnings produced by the analyzers, if they are dominated by documentation and formatting issues. This factor should be considered when analyzers are used in automatic evaluator systems.

3. Methodology

The first batch containing 3433 solutions written in C++ were collected from the *Object-oriented programming* course. While the students have to develop command-line interface applications, they have to manage memory manually and use advanced object-oriented techniques, like polymorphism. In addition, 226 C++ projects were added from the *GUI programming with QT* course, where the participants have to develop complex graphical application with layered architecture (Model-View).

The C++ submissions were analyzed with *Clang-Tidy*, *Clang Static Analyzer* and *Cppcheck* [1, 13, 15, 16]. To run the previously mentioned tools and visualize their results, we have used *Ericsson CodeChecker* [8].

In the case of C# projects, 2148 programming submissions were collected from the *Event-driven programming* course, where students have to develop Windows Forms, WPF and Xamarin/MAUI graphical applications. Similarly to the *GUI programming with QT* course, the usage of layered architecture (Model-View and Model-View-ViewModel) is mandatory. For these programs, we have used both first-party (*Microsoft NetAnalyzers*) and third-party (*Roslynator Analyzers* and *SonarAnalyzer CSharp*) analyzers built on top of APIs provided by the *Microsoft Roslyn* compiler platform [23, 24].

Table 1 summarizes the previously described tools, courses, and analyzed submissions.

Language	**Analyzer tools**	**Course**	**Submissions**
C++	Clang Tidy, Clang Static Analyzer, Cppcheck	Object-oriented programming	3433
		GUI programming with QT	226
C#	Microsoft NetAnalyzers, Roslynator Analyzers, SonarAnalyzer CSharp	Event-driven programming	2148

Table 1. Summary of the used analyzers and evaluated submissions

From the findings of the analyzers, we have selected the presented errors according to the following criteria:

- We have included the most common and typical errors.
- Some errors only occurred in a handful of submission, but they indicated serious design flaws or lack of understanding.
- We excluded styling errors. While code-styling is important, there were no enforced styling guidelines for the assignments. Also, these rules often require detailed configuration in real-world projects.
- For the C# programs a significant part of findings reported by the Roslyn-based analyzers were possible refactorings, those were also discarded.

4. C++ results

In this section, we present the selected errors from the C++ solutions. Figure 1 shows the number of solutions from both courses where the those errors occurred.

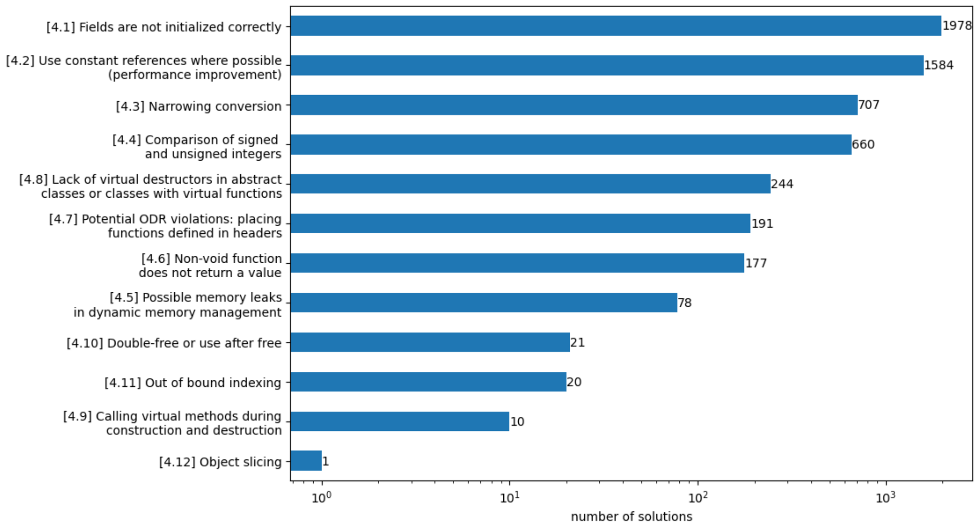

FIGURE 1. The number of C++ solutions with errors

4.1. **Fields are not initialized correctly.** Fields are usually not initialized automatically in C++. However, they could be initialized during debug compilation on some platforms, misleading students (as in Listing 1). It is generally a good practice to give sensible starting values to fields during construction.

```
class Example {
private:
    int _x, _y, _z;
public:
    Example(int x, int y): _x(x), _y(y) {}
    void method() {
        if (_z > 0) { /* ... */ }
        // _z not guaranteed to be initialized to zero
    }
};
```

LISTING 1. Field initialization

4.2. **Use constant references where possible (performance improvement).** While function `f` in Listing 2 is functionally correct, it has two potential performance problems:

(1) the `vec` vector is copied every time `f` is called;
(2) the `curr` vector is copied in every iteration.

Using `const` references for the parameter and the loop variable improves performance of this program.

```
void f(vector<vector<int>> vec) {
    for(vector<int> curr : vec) { /* ... */ }
}
void f_improved(const vector<vector<int>>& vec) {
    for(const vector<int>& curr : vec) { /* ... */ }
}
```

Listing 2. Perfomance can be improved with constant references

However, sometimes copying the values of the parameters is the desired behavior. Code analyzers are smart enough to give hints based on the context. So, these warning are only showed if marking the parameter to a `const` reference would not break the given program.

4.3. **Narrowing conversion.** Conversion from a wider data type to a narrower can lead to data loss (e.g., `float` → `int`) and/or integer overflow (e.g., `unsigned int` → `int` in Listing 3).

```
void search(int elem, bool& found, int &ind) {
  found = false;
  for (unsigned int i = 0; i < vec.size() && !found; ++i) {
    if(vec[i] == elem) {
      found = true;
      ind = i; // Could cause integer overflow
    }
  }
}
```

Listing 3. Possible integer overflow, because of narrowing conversion

4.4. **Comparison of signed and unsigned integers.** Direct comparison between signed and unsigned integers is not safe in C++. In most cases this error occurred, when students compared a loop-variable with a size of a container that has `std::size_t` type which is an unsigned integer type (Listing 4). While this have not caused problems in the submitted solution, it is still considered a bad practice, because `vec.size()` can be greater than the maximum value of `int` on the given platform.

```
for (int i = 0; i < vec.size() /* std::size_t */ ; ++i) {}
```

Listing 4. The maximum size of `int` might be smaller than `vec.size()`

Comparison of signed and unsigned integers could also be problematic if the signed integer value is negative. In Listing 5, we would expect that it will print `0` as *i* is not greater than *j*, but the value of *i* is also cast to `unsigned int` and it underflows.

```
int i = -4;
unsigned int j = 5;
std::cout << (i > j) << std::endl; // Expected 0, but prints 1
```

Listing 5. `int i` is casted to `unsigned int`

4.5. **Possible memory leaks in dynamic memory management.** Freeing allocated dynamic memory is often missed by students. Consider the `Stack` class in Listing 6, where the writer of the code allocates memory for the array, but the destructor is missing. Thus, the memory will not be freed after *s* is not used anymore.

```
class Stack {
private:
  int _top, _size;
  int* _arr;
public:
    Stack(int size)
    : _top(-1), _size(size), _arr(new int[size]) {}
    // ...
};
```

Listing 6. Memory leak: missing destructor

4.6. **Non-void functions does not return a value.** Reaching the end of the body of a non-void function without returning a value is will not generate a compiler error by default, but it is an undefined behavior in C++. A good example of this, a stack class where the `pop` method of a stack that removes the item from the top of the stack and returns its value. The implementation in Listing 7 of the `pop` method is error-prone, because the user of the class can call the method on an empty stack.

```
int Stack::pop() {
    if (!isEmpty()) { return _vec[--_top]; }
}
```

Listing 7. Empty stacks are not handled

4.7. **Potential ODR violations: placing function in headers.** One Definition Rule (ODR) means that non-inline functions and types must have only one definition in the entire program [21]. For instance, placing functions in headers can lead to ODR violations. This does not necessarily mean that the solution does not compile or run until it is only included in one source file. However, if the student had included it in two or more sources, the compiler would not have accepted the solution.

Consider the scenario illustrated in Listing 8, while

- the `g++ main.cpp first.cpp` command will compile the program successfully;
- the `g++ main.cpp first.cpp second.cpp` command will fail.

```
/// Contents of helpers.h:
int square(int x) { return x * x; }

/// Contents of first.cpp:
#include "helpers.h"
void first_calculation() { int res = square(2); /* ... */ }

/// Contents second.cpp:
#include "helpers.h"
void second_calculation() { int res = square(3); /* ... */ }

/// Contents main.cpp:
// helpers.h is not included in main.cpp
```

Listing 8. Functions in headers

4.8. **Lack of virtual destructors in abstract classes or classes with virtual functions.** It is possible that the student implemented all necessary destructors, but they are not marked as virtual when needed. In Listing 9, if the destructor of `Base` is not marked as virtual and `delete` is called on a variable with static type of `Base`, then the destructor of `Derived` will not be called.

```
struct Base {
    virtual void method() = 0;
    ~Base() {std::cout << "base "; }  // Should be virtual
};
struct Derived: public Base {
    void method() override {  }
    ~Derived() { std::cout << "derived "; }
};
void f() {
    Base* d = new Derived;
    delete d; // outputs: base
}
```

Listing 9. Destructors should be `virtual`

4.9. **Calling virtual methods during construction and destruction.** During construction and destruction, the virtual call mechanism is disabled. Therefore, the implementation from the current class is used, as illustrated in Listing 10 with the call of `f`. Calling virtual methods in the constructor is not necessarily a problem, but the student might not aware of the previously described behavior.

```
struct Base {
  Base() { f(); } // Prints base
  virtual void f() { std::cout << "base"; }
};
struct Derived: public Base {
  void f() override { std::cout << "derived"; }
};
```

Listing 10. Virtual calls in constructors

4.10. **Double-free and use after free.** In C++ `delete` should be called only once for the same reference and the reference should not be used after `delete` called on it. Listing 11 counts not only as double-free, but an infinite loop, because `~Example` will always get called again, recursively. This is a good example of how reported errors can also indicate lack of understanding from students.

```
struct Example {
    ~Example() { delete this; }
};
```

Listing 11. Incorrect usage of `delete`

4.11. **Out of bound indexing.** Out of bound indexing is often missed by beginner programmers. It usually results in a *memory segmentation fault*. The provided example (Listing 12) is relatively simple: the student made a small mistake and wrote `<=` instead of `<`. Fortunately, the used code analyzers can spot possible out of bound indexing in more complex scenarios.

```cpp
int arr[10];
for (int i = 0; i <= 10; ++i) { arr[i] = i; }
```

Listing 12. Out of bound indexing

4.12. **Object slicing.** Slicing happens when copying a derived object into a base object: the members of the derived object (both member variables and virtual member functions) will be discarded [5]. In Listing 13, slicing object from type `Derived` to `Base` discards override `method`.

```cpp
struct Base {
    virtual void method() { std::cout << "base"; }
};
struct Derived: public Base {
    virtual void method() { std::cout << "derived"; }
};
void f(Base obj) { obj.method(); }
int main() {
    Derived d;
    f(d); // prints base
    return 0;
}
```

Listing 13. Object slicing

5. C# results

In this section, we present the selected errors from the C# solution. Figure 2 shows the number of solutions from both courses where the those errors occurred, categorized by tasks. It is worth to note that the number of solutions containing the highlighted errors are really similar for the WinForms, WPF, and Xamarin/MAUI assignments. This is because the students have to develop the same software for all three tasks, and they are encouraged to reuse layers from their previous solutions. Exams are different, because student have to develop new applications from scratch, but reusing their existing materials is still allowed.

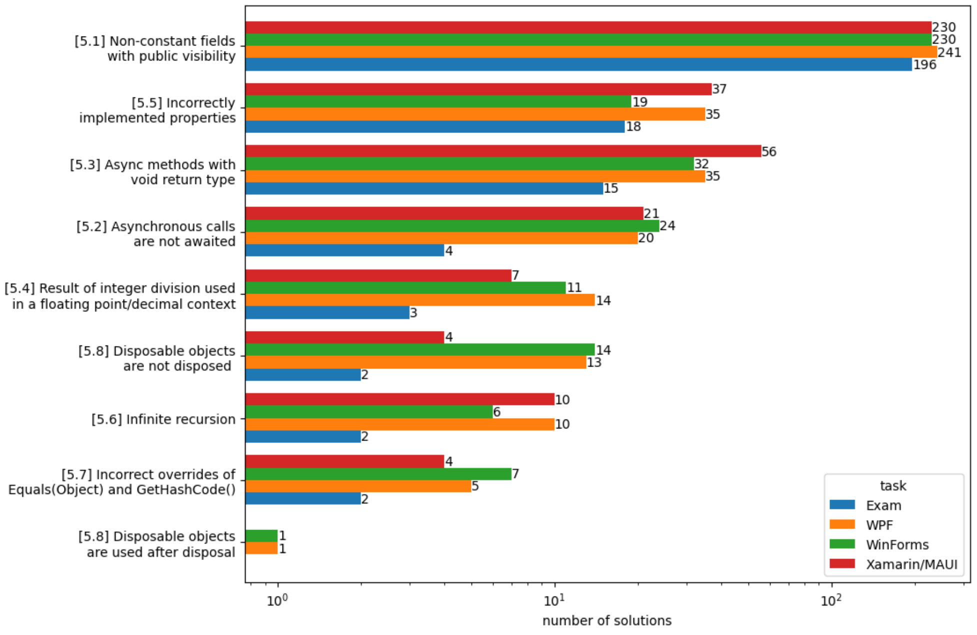

FIGURE 2. The number of C# solutions with errors

5.1. **Non-constant fields with public visibility.** Using public mutable fields are generally considered a bad practice and against guidelines in C#. There are several alternatives:

- mark the field `readonly` or `const`;
- use auto-implemented properties instead;
- make it private and access it with a property or method.

5.2. **Asynchronous calls are not awaited.** In Listing 14, `NewGameAsync` is an async function, but it is not awaited. Thus, the state of the `model` object might be incorrect when `AdvanceGame` is called.

```
GameModel model = new GameModel();
model.NewGameAsync();
model.AdvanceGame();
```

LISTING 14. `model.NewGameAsync()` is not awaited

5.3. **Asynchronous methods with void return type.** Asynchronous function should return `Task` or `Task<T>`, because they cannot be awaited and exceptions cannot be caught from them (Listing 15). Event handlers are the only exceptions according to the Microsoft Learn guidelines, because they usually have to return `void` [17].

```
public async void LoadAsync(string filePath) {
    FileContent = await File.ReadAllTextAsync(filePath);
}
```

LISTING 15. `LoadAsync` cannot be awaited

5.4. **Results of integer division should not be assigned to floating point/decimal variables/parameters.** In Listing 16, if the result of `size / 2` is positive, then it is already floored because of the integer division. Therefore, calling `Ceiling` will not return the expected result.

```
int size = // ...
if ((int)decimal.Ceiling(size / 2) == x) { /* ... */ }
```

LISTING 16. Integer disivision

5.5. **Incorrectly implemented properties.** The getters and setters of the properties should access the correct backing fields. As shown in Listing 17, the student may want to write a read-only property, but the setter is still present with an empty body. The correct solution would be a property without a setter, because assigning a value to a property will not generate a compile-time error and the user may think that the property is writable. In contrast, if the setter is not present, then both the compiler and the IDE will show an error upon assignment.

```
private int property;
public int Property { get { return property; } set {} }
```

LISTING 17. Read-only property: `set` should be omitted

5.6. **Infinite recursion.** A trivial example of this error, when the setter tries to assign the vale to the property itself (Listing 18). This may be the result of a typo in the source code, as backing fields often have the same name as the property, but with a different case. Calling the setter of such a property would lead to an infinite recursion.

```
private int property;
public int Property {
    get { return property; }
    set { Property = value; }
}
```

LISTING 18. Incorrectly implemented setter: infinite recursion

5.7. Incorrect overrides of Equals(object) and GetHashCode(). When overriding `Equals(object)` and `GetHashCode()` certain rules should be followed, such as:

- `Equals(object)` and `GetHashCode()` should be overridden in pairs.
- Classes directly extending `object` should not call `base` in `GetHashCode` or `Equals`. The implementation in `object` are based on object reference.

In Listing 19, the student overrides both methods, but the `GetHashCode` calls the implementation from the `object` class.

```
class ClassName {
    public override int Equals() {
        // correct implementation
    }
    public override int GetHashCode() {
        base.GetHashCode(); // Calls GetHashCode from object
    }
}
```

LISTING 19. Incorrect override of `GetHashCode()`

5.8. Mistakes related to disposable objects. While C# has automatic garbage collector, the unmanaged resources taken by certain classes should be freed. For instance (Listing 20), if a file opened by the `StreamReader` class, then it should be closed after usage. The `Dispose` should be called (or `Close` in this case), preferably in a `finally` block. A `using` statement or declaration would be an even better option, as it ensures the correct usage of disposable objects.

```
List<string> values = new List<string>();
StreamWriter sw = new StreamWriter("output.txt");
foreach (string line in values) { sw.WriteLine(line); }
```

LISTING 20. The file is not closed after usage

It is also important that the objects cannot be used after they are disposed. In the example in Listing 21, the `Dispose` method is automatically called after the execution leaves the block of the using statement. So, the returned `StreamReader` instance will not be able to read the file, as its methods will throw an `ObjectDisposedException`.

```
public StreamReader CreateReader(string filename) {
    using (StreamReader sr = new StreamReader(filename)) {
        return sr;
    }
}
```

Listing 21. `sr` is returned after disposal

6. Integration with automated evaluator systems

We developed a prototype-implementation as part of the open-source *TMS* task management system developed at ELTE FI [7], which already contains a custom developed Docker-based automated evaluator and integrates the static analysis tool *CodeCompass* [20], but for code comprehension purposes.

We extended the evaluator system with the tools mentioned in Section 3. *CodeChecker* is an important part of our solution, because apart from running static analysis on C/C++ solutions with *Clang SA*, *Clang Tidy*, and *Cppcheck* it can also process the output of more than 20 third-party analyzers and convert it to its own format. Thus, it is enough if TMS can process only one report format. Moreover, we could take advantage of the additional tools provided by *CodeChecker*, such as the HTML report viewer.

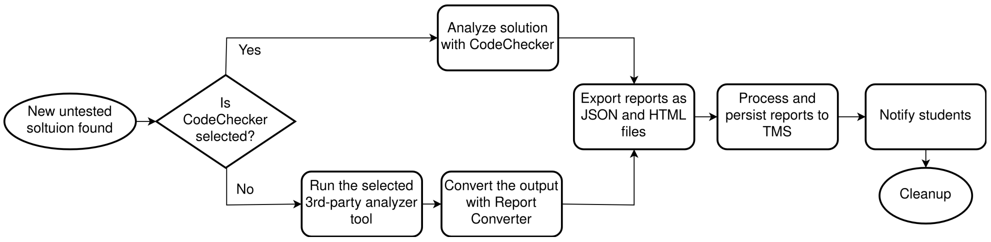
New untested soltuion found
Is CodeChecker selected?
Yes
No
Analyze solution with CodeChecker
Run the selected 3rd-party analyzer tool
Convert the output with Report Converter
Export reports as JSON and HTML files
Process and persist reports to TMS
Notify students
Cleanup

Figure 3. The workflow of automated static code analysis in TMS

The Docker environment and the selected tools can be individually configured for each assignment. So, instructors can adapt the tools to the need of their courses. The system regularly checks for new submissions from tasks with valid evaluator configuration. When there is a new untested solution, the system runs the selected tools in Docker containers and converts the reports to a common report format if necessary. Finally, when the reports are available in the required format, TMS persists them for the given solution, and notifies the student about the completion of the analysis. The described workflow is illustrated in Figure 3.

Both the students and the instructors can view the results on the page of the given submission. It is important that the results should be presented in a legible format. They are prioritized by severity to help students identify the more serious issues first. The HTML reports produced by CodeChecker are also available from the user interface, so the reports can be viewed without downloading the submissions.

7. Conclusions and Future Work

In our paper, we have evaluated over 5000 student submission written in C++ and C#, by running static code analyzer tools on them. We have found violations of conventions and various programming bugs which could have been filtered with static analysis, but were overlooked by the teachers, probably due to the high number of student submissions they had to evaluate and grade. In these cases, the feedback provided by the analyzers could help students to fix their mistakes before the deadlines and learn from them. Furthermore, the usage of these tools would allow a more thorough assessment by teachers and speed up the grading process.

While analyzing solutions from previous semesters helped us to create the initial prototype implementation, choose the right tools, and configure them, we believe our solution can be improved further by testing it during an academic semester. First, it should be observed what is the impact of such a tool on the students' behavior, how much the quality of their submissions improved. Furthermore, it should also be determined if the students really understand and use correctly the provided feedback. The current implementation shows the reports to both students and instructors, but some tips might be applied too easily without understating them. It might be beneficial to make the detail of the feedback configurable or add an option to hide it from the students completely, so instructors can choose according to their preferences. Another possible approach could be adding an option to limit the number of possible uploads, thus students have to rethink twice before re-upload their solutions just to check if they managed to solve the reported problems. Finally, after the testing is completed and the previous questions are issued, we aim to introduce our solution for other courses at ELTE FI.

References

[1] Babati, B., Horváth, G., Májer, V., and Pataki, N. Static analysis toolset with Clang. In *Proceedings of the 10th International Conference on Applied Informatics* (2017), pp. 23–29.

[2] Bardas, A. G., et al. Static code analysis. *Journal of Information Systems & Operations Management 4*, 2 (2010), 99–107.

[3] Birillo, A., Vlasov, I., Burylov, A., Selishchev, V., Goncharov, A., Tikhomirova, E., Vyahhi, N., and Bryksin, T. Hyperstyle: A tool for assessing the code quality of solutions to programming assignments. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (New York, NY, USA, 2022), SIGCSE 2022, Association for Computing Machinery, pp. 307—313.

[4] Blau, H., and Moss, J. E. B. Frenchpress gives students automated feedback on java program flaws. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2015), ITiCSE '15, Association for Computing Machinery, pp. 15—-20.

[5] Clang Team. LLVM - Clang-tidy - cppcoreguidelines-slicing. https://releases.llvm.org/13.0.0/tools/clang/tools/extra/docs/clang-tidy/checks/cppcoreguidelines-slicing.html, Accessed: 2023-02-25.

[6] Edwards, S. H., Kandru, N., and Rajagopal, M. B. Investigating static analysis errors in student java programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (New York, NY, USA, 2017), ICER '17, Association for Computing Machinery, pp. 65—-73.

[7] ELTE. TMS – Task Management System. https://tms-elte.gitlab.io/, Accessed: 2023-02-27.

[8] Ericsson Ltd. CodeChecker. https://codechecker.readthedocs.io/, Accessed: 2023-02-25.

[9] Eurostat. ICT education - a statistical overview. https://ec.europa.eu/eurostat/statistics-explained/index.php?title=ICT_education_-_a_statistical_overview, Accessed: 2023-04-16.

[10] Gomes, I., Morgado, P., Gomes, T., and Moreira, R. An overview on the static code analysis approach in software development. *Faculdade de Engenharia da Universidade do Porto, Portugal* (2009).

[11] Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (New York, NY, USA, 2010), Koli Calling '10, Association for Computing Machinery.

[12] Keuning, H., Heeren, B., and Jeuring, J. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2017), ITiCSE '17, Association for Computing Machinery, pp. 110—-115.

[13] Kremenek, T. Finding software bugs with the clang static analyzer. *Apple Inc* (2008).

[14] Loyalka, P., Liu, O. L., Li, G., Chirikov, I., Kardanova, E., Gu, L., Ling, G., Yu, N., Guo, F., Ma, L., Hu, S., Johnson, A. S., Bhuradia, A., Khanna, S., Froumin, I., Shi, J., Choudhury, P. K., Beteille, T., Marmolejo, F., and Tognatta, N. Computer science skills across china, india, russia, and the united states. *Proceedings of the National Academy of Sciences 116*, 14 (2019), 6732–6736.

[15] Marjamäki, D. Cppcheck. https://cppcheck.sourceforge.io/, Accessed: 2023-02-23.

[16] Martignano, M., and Spazio, I. A new static analyzer: The compiler. *ADA USER 40*, 2 (2019), 99–103.

[17] Microsoft. Async return types (C#). https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-return-types, Accessed: 2023-02-23.

[18] Molnar, A.-J., Motogna, S., and Vlad, C. 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* (New York, NY, USA, 2020), EASEAI 2020, Association for Computing Machinery, pp. 7—12.

[19] Orr, J. W. Automatic assessment of the design quality of student python and java programs. *arXiv e-prints* (2022).

[20] Porkoláb, Z., Brunner, T., Krupp, D., and Csordás, M. Codecompass: an open software comprehension framework for industrial usage. In *Proceedings of the 26th Conference on Program Comprehension* (2018), pp. 361–369.

[21] Quinlan, D., Vuduc, R., Panas, T., Haerdtlein, J., and Saebjoernsen, A. Support for whole-program analysis and the verification of the one-definition rule in C++. In *Static Analysis Summit 2006* (6 2006).

[22] Striewe, M., and Goedicke, M. A review of static analysis approaches for programming exercises. In *Computer Assisted Assessment. Research into E-Assessment* (07 2014), vol. 439, Springer, pp. 100–113.

[23] Sundström, J. Assessment of Roslyn analyzers for Visual Studio, 2019.

[24] Vasani, M. *Roslyn Cookbook.* Packt Publishing Ltd., 2017.

ELTE Eötvös Loránd University, Faculty of Informatics, Department of Software Technology and Methodology, H-1117 Budapest, Pázmány P. sny 1/C.

Email address: `t5mop2@inf.elte.hu`

Email address: `mcserep@inf.elte.hu`