Hindawi
Complexity
Volume 2023, Article ID 9876801, 1 page
https://doi.org/10.1155/2023/9876801

Retraction

Retracted: Automatic Grading for Complex Multifile Programs

Complexity

Received 19 December 2023; Accepted 19 December 2023; Published 20 December 2023

This article has been retracted by Hindawi following an investigation undertaken by the publisher [1]. This investigation has uncovered evidence of one or more of the following indicators of systematic manipulation of the publication process:

(1) Discrepancies in scope
(2) Discrepancies in the description of the research reported
(3) Discrepancies between the availability of data and the research described
(4) Inappropriate citations
(5) Incoherent, meaningless and/or irrelevant content included in the article
(6) Manipulated or compromised peer review

The presence of these indicators undermines our confidence in the integrity of the article's content and we cannot, therefore, vouch for its reliability. Please note that this notice is intended solely to alert readers that the content of this article is unreliable. We have not investigated whether authors were aware of or involved in the systematic manipulation of the publication process.

Wiley and Hindawi regrets that the usual quality checks did not identify these issues before publication and have since put additional measures in place to safeguard research integrity.

We wish to credit our own Research Integrity and Research Publishing teams and anonymous and named external researchers and research integrity experts for contributing to this investigation.

The corresponding author, as the representative of all authors, has been given the opportunity to register their agreement or disagreement to this retraction. We have kept a record of any response received.

References

[1] T. Wang, D. B. Santoso, K. Wang, and X. Su, "Automatic Grading for Complex Multifile Programs," *Complexity*, vol. 2020, Article ID 3279053, 15 pages, 2020.

Hindawi
Complexity
Volume 2020, Article ID 3279053, 15 pages
https://doi.org/10.1155/2020/3279053

Research Article

Automatic Grading for Complex Multifile Programs

Tiantian Wang,[1] Djoko Budi Santoso,[1] Kechao Wang,[2] and Xiaohong Su[1]

[1]*School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China*
[2]*School of Information Engineering, Harbin University, Harbin 150001, China*

Correspondence should be addressed to Tiantian Wang; wangtiantian@hit.edu.cn

Received 31 March 2020; Accepted 23 May 2020; Published 13 June 2020

Guest Editor: Zhihan Lv

This paper presents an automatic grading method DGRADER, which handles complex multifile programs. Both the dynamic and the static grading support multifile program analysis. So, it can be an advantage to handle complex programming problem which requires more than one program file. Dynamic analysis takes advantage of object file linker in compilation to link complex multifile program. The static grading module consists of the following steps. Firstly, the program is parsed into abstract syntax tree, which is mapped into abstract syntax tree data map. Then, the information of preprocessor is used for linking external sources called in main program by complex multifile program linker-fusion algorithm. Next, standardization process is performed for problematic code removal, unused function removal, and function sequence ordering based on function call. Finally, program matching successfully tackles structure variance problem by previous standardization process and by simple tree matching using tag classifier. The novelty of the approach is that it handles complex multifile program analysis with flexible grading with consideration of modularity and big scale of programming problem complexity. The results have shown improvement in grading precision which gives reliable grading score delivered with intuitive system.

1. Introduction

Automatic Grading System (AGS) is program which can determine student grade automatically based on score objective parameter. AGS is needed as popular courses especially in computer science often have hundreds or thousands of students but only a few staff [1]. Programming assignment in these courses is necessary to improve technical programming and problem solving skill of students. Manual assessment method of programming practices is a tedious and time-consuming task [2, 3]. This method is inadequate because first programming course are often typically complex for a lecturer to assess correctness accurately and comprehensively by manual assessment [4]. In here, AGS can become a key role to maintain accuracy and avoid biases as the grading process is based on objective scoring rules. Importantly, the system must provide immediate feedback to students so they can learn from their mistakes [5], allowing them to make self-learning without an instructor [6].

With the development of information technology, more and more schools and organizations try to realize the inclusive and fair education through online learning, human-computer interaction [7, 8] learning, etc. Nowadays, various AGS systems already exist and are used by higher education institutions to enhance learning process [9, 10] or as programming communities to self-improve in programming and problem solving skills. The question "why do so many automatic assessment systems exist, and why are new ones created every year?" pops up into discussion in this research field [11]. One of the reasons for various AGS is that each system may deliver some features which are not provided by others to distinguish their advantage factor.

The research still continues until today to pursue reliable grading system and solve the issues existed in this field. In here, current major issues are listed, which work as a sample. This gives an idea to produce final reliable grading and potential future work.

According to Table 1, these are recent knowledge issues chosen as the main objective overview in this work to do novelty works.

Firstly, code variance which can be said as the biggest factor to affect the result of static program analysis. In static

Table 1: Current research issues in automatic grading system.

Issue	Approach	Results
Code variations are widely believed to impede program analysis because various source codes have to be recognized as the same [12]	Program normalization by using system dependence graph (SDG) implemented in prototype called normalizer	Successful for small-sized program, not big-sized or complex program
Most existing inlining algorithms not suitable for code analysis and interprocedural analysis can analyze the calling context but are very costly and cannot remove function call variations [13]	Inlining algorithm based on program dependence graph (PDGs) using simple function call tree (SCFT) applied to code normalization	Limited in dealing with programs with multiple files
Deals with multiplicity of solutions that exists for the same programming problems providing automated evaluation: immediate feedback for students and grading assignment of program [14]	Partitioned block by using control flow graph (CFG) and program behavior comparison with symbolic execution rather than textual context or concrete values	Solves a multitude of solution of problems but is complex in implementation for big scale program

analysis, these issues must be considered to produce reliable and precise grading score. Code variance issues solved by introduced program normalization mechanism to change the program into an intended structure requiring a program representation. The program normalization approach uses system dependence graph as the program representation. It requires for changing the code or program component known as refactoring process. The results of the works that solve code variance issue have significant impact of grading quality result.

Secondly, state that current inlining algorithms are not suitable for code analysis. The work approach by using program representation in Program Dependence Graphs to expand and manipulate program component focused on function call is introduced. This issue is actually related to code variance and focused on the structure program, which is influenced by function call and function sequence order written in program code. Results of the approach give more suitable inlining algorithm for code analysis and improve mechanism by reducing step of conventional interprocedural analysis. However, the existing work cannot deal with functions in several files.

Thirdly, Arifi et al. have introduced how the program is partitioned by a block using Control Flow Graph and generated symbolic variance for program comparison to solve the problem of multiplicity of solutions [14]. This work solves a multitude of solution of problems, but is complex in implementation for big scale programs.

These research studies lay a good foundation for our work. However, further work is still needed to support complex multifile programs. As the current single file program analysis is a relatively straight forward and common, complex multifile program analysis is a plus to handle big complex program which is still not solved or suitable for the recent research [15]. In this paper, we present a novelty work to handle complex multifile program analysis.

2. Related Work

2.1. Complex Multifile Program Analysis. AGS should be capable to quickly grade multiple and complex computer literacy assignments while providing meaningful feedback in order to stimulate an efficient learning process [16]. Specially, for larger and complex programming assignment, it is still not possible for most existing systems to assess good programming solution. In this case, semiautomated system is still used, which requires the human evaluator to use partial part of results from system to decide final grading [17]. It makes the task more complicated for humans to assess modular code. This issue will be faced by the current grading system as a programming problem is more scaled and complex for advance programming courses or real practice programming project. It makes the automatic grading system not follow their primary principle as an automatic system and become ineffective.

Complex multifile program analysis can be an advantage to handle complex programming and scaled project, which requires more than one file program. As stated before, it also makes modularity and flexibility such as creating user-based custom libraries to support core program file without rewriting into main program. In this feature, the implementation of complex multifile program analysis uses their primary concept.

2.2. Dynamic and Static Grading Methods. Dynamic analysis uses black-box concept which depends on output results. The analysis requires program to be compiled and run with test cases. The final grade uses comparison results of the produced output with the expected output. However, it has fatal drawback, in which student program may not produce an output because problematic code such as syntactical errors makes the program fail in the compilation process [18]. This means the program will fail before processing test cases. Hence, dynamic analysis itself is not enough for completely giving all objective scores when it fails and static analysis is needed.

Static analysis uses white-box concept without compiling and running the program. It uses rapid advancement technology from compiler and language-based tool. The approach also uses code analysis knowledge to analyze program and gives grading based on scoring objective rules and parameters.

However, both methods are having their advantage and disadvantage since static analysis require more complex process of code analysis. It requires predefined rules and objectives. On the contrary, dynamic analysis is a more direct approach but does not cover all aspect, especially, when it fails to perform grading. Choosing which methodology is more feasible makes dynamic versus static analysis become a topic for consideration for the grading system. According to dynamic and static analysis principle in the previous section, both methods have their advantage and disadvantage factors. Table 2 summarizes of their comparison.

In here, the knowledge summarizes that static analysis cannot be used for checking the correctness of student programs using test cases as an input which produces an output. On the contrary, traditional dynamic analysis systems will completely fail to perform grading and miss important aspects when assessing student programs such as checking the code quality [19]. This may be the reason to why some existing automatic grading systems combine the best of both approaches by improving dynamic testing mechanism with static technique. By carrying out mutual combination of both analysis and providing immediate feedback in grading result, it gives an additional positive value to the user and advantage of the grading system.

Back to 1992, when Cellidh [20] was introduced, it was actually a pioneer grading system which combined both approaches by introducing semantic error detection. It used for detecting infinite loop issue which is critical for dynamic grading. The system also uses static verification including structure, indentation, detect comment, readability measurement, and complexity metrics in dynamic analysis.

In 1997, system called ASSYST [21] combined both analyses in practice to automate some aspects of grading for introductory Ada classes, as well as a second-year C programming course. It gives grading score to students based on the correctness (actual output compared to the expected output), efficiency (run time) on dynamic analysis side, check program source code style, and its complexity on static analysis side. In 2000, more systems adopt this combined method for providing flexible analysis and pursuing potential grading system which provide meaningful intermediate feedback.

In 2006, Marmoset [22] was built in the University of Maryland. The main purpose of the system is to collect information about development of student programming skill while doing a programming assignment for triggering self-improvement. The unique feature is allowing a full snapshot about student progress in the system, so it can be analyzed in detail by using different types of test cases (student, public, release, and secrete) and a personal support from the lecturer through comments on the code in the page.

In 2008, Web-CAT [23] provided extensibility and flexibility as its plugins-based architecture taking advantage of recent development technologies was built using Java servlet. It provides security features by authentication, erroneous or dangerous code detection, and portability. It also supports manual grading by allowing the lecturer to check program submitted by the student. It allows lecturers to give the comment, suggestion, and grading modification. The programming language supported are C or C++, Java, Pascal, Prolog, and others flexibility support for integration. The grade is based on correctness through test cases, completeness of program, and validity.

In 2011, eGrader [24] provided detailed feedback reports and allowed students to see model solution provided by the lecturer or course owner. It also gives specific comments on syntax and semantic errors if occurred. The static analysis process implemented in the system consists of two parts, which are the structural similarity and quality analysis. Structural similarity analysis is based on the graph representation of the program. Quality analysis was achieved by measurements using software metrics.

In the same year, a system called AutoLEP [25], as an automatic grading system tool, was developed. It improves the traditional static grading mechanisms by combining dynamic code testing. The approach is enriching static analysis in source code analysis with a comparison of the similarity degree of compared program. The dynamic analysis was used to evaluate correctness of the submitted program using test cases and comparing the expected output. The static analysis does not compile or execute the programs. It uses model program to evaluate student program construction and how close the student source code is from the correct solution which is model programs provided in programming assignment by the lecturer. The final grading result was achieved by calculation of summarization from each grading analysis. The works were reported to distinguish syntactic and semantic analysis from the previous work. The architecture includes (1) the client and a computer used by a student, and it performs the static analysis and can provide quick feedback; (2) a testing server which has to perform the dynamic analysis; and (3) a main server which has to control the information of the other components to establish a grade.

In 2012, a new automatic grading system called Quimera [26] was built as a web-based application. It was able to evaluate the program source code written in C language and provide a full management system for programming contests. It also allows to create and manage programming exercises both in competitive learning and programming contest environments. Besides the traditional dynamic approach, this system provides a static analysis of the program by measuring the source code quality. Thus, the final grade is based not only on the source code capability of producing the expected output but also on its quality and accuracy.

Finally, these are examples of automatic grading tools existed with flexible code analysis which primarily combine both method to achieve advantages of grading in their grading system. The listed tools will be used in comparison analysis.

3. Overview

As mentioned before, our approach is implemented in our automatic grading system tools called DGRADER as a web-based online automatic programming judgement platform. Web platform chosen as a researcher focused on web-based assessment system shows positive influence on learning effectiveness [27]. It is also very effective in distributing material and collecting the student assignment online [28].

Table 2: Dynamic versus static analysis in the automatic grading system.

Method	Concept	Advantage	Disadvantage
Dynamic	Black-box (output) with test cases	(i) More direct for grading (ii) Correctness checking (iii) Popular usage	(i) Requires compilation process and fails when compilation is unsuccessful (ii) Security issues (iii) Does not cover all aspect of grading
Static	White-box (code analysis) and comparison of the correct model	(i) Does not require compilation process (ii) Capable to analyze code quality	(i) Computational complexity (ii) Requires set of rule definitions for giving reliability grading (iii) Not providing correctness checking

Our tool is built by using Spring Model View Controller (MVC) as framework foundation with Spring Tool Suite (STS) Integrated Development Environment (IDE) written in Java programming language. It uses several technologies integrated to the system to support MOOCs. The full system will be running on the host server which can support Apache Tomcat Server. The database technology used in this application is MySQL database.

Architecture: Figure 1 shows three blocks of the system, i.e., user, application (front and back-end), and expanded core grading API block. Every core page contains a block of the modular page with defined web services and API. The system manages basic web features provided by Spring such as servlet, session handler, and its core features.

Core grading Application Programming Interface (API) of the system is shown on the right side. It has several API for specific purposes and functions. The API is triggered by using defined query parameter and handled by the API handler. Core grading APIs cover main functions of DGRADER for the grading task. The first layered API is the assignment handler. Its main purpose is to handle raw materials of assignments and extract the information before the grading task. Submission API is the connector triggered by submission activities. Its role is to forward information to grading analysis process in the grading analysis APIs. The information passed by the query parameter is used to target specific grading in the system. Each grading will be detailed in its section. Finally, the grading activities will produce an output feedback which will be displayed in the interface to users.

3.1. Main Features. The main features of DGRADER are as follows:

(1) Providing e-learning environment management for programming courses.

(2) Supporting flexible grading analysis: dynamic and static analysis. Flexible grading makes the instructor or programming assignment task creator become more flexible to choose assessment methods. The grading method can be based on the complexity of programming task or its purpose. It also makes the system capable to cover grading assessment task when one of the grading fails.

(3) Supporting complex multifile program analysis in submission activity for solving one programming problem.

(4) Providing instant feedback and result of grading analysis for programming assignment problem. The grading is provided in both categorical and numerical results.

Feedback results in static analysis include

(i) Presenting the final linked-fused program source code and its standardization result

(ii) Presenting AST traverse log with visitor activities

(iii) Providing visualization for transformation graph comparison for final linked-fused and standardized program structure

(iv) Presenting function information such as function list in program, function call sequence, and unused function list removed by standardization

(v) Presenting the original AST data mapper of final program and standardized AST data mapper of standardized program

(vi) Providing root tree visualization by AST root data mapper

(vii) Providing experimental AST visualization of standardized program by standardized AST data mapper

(viii) Presenting AST simple tree matching trace analysis in data mapper

(5) Flexible source program uploading or submission by using file upload or directly using integrated CodeMirror text editor.

(6) Supporting course system management similar to Massive Open Online Courses (MOOCs) and Learning Management System.

4. Multiple Program Files Analysis

Multiple program file analysis is a novel feature of our automatic grading system. The reason to present this feature is because existing systems usually support single file to keep simplicity of its grading analysis process. However, if a programming problem is more complex and users need more files, this will become a limitation factor. In other cases, users may want to use some libraries which do not require to be written again or the compiler in dynamic analysis does not support the library. In order to support multiple program file submission, this section presents implemented approaches including file model, multiple program file linker in each grading process, and linker-fusion algorithm.

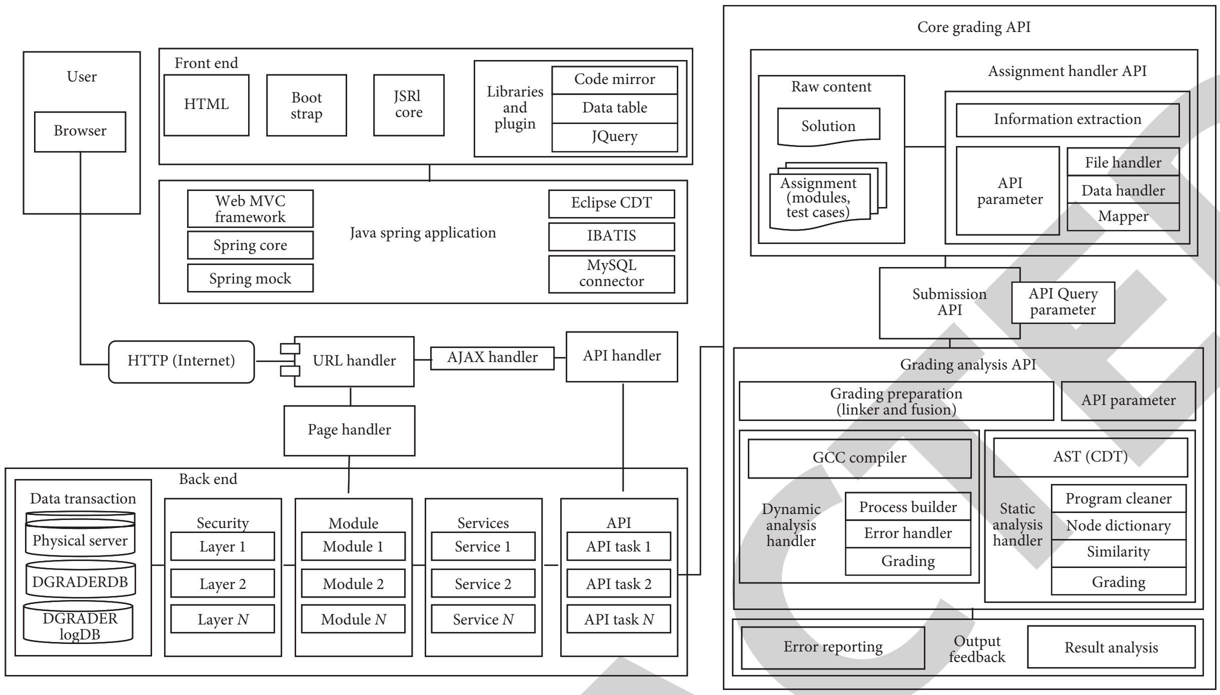


Figure 1: System architecture.

4.1. File Model. File model is implemented to handle multiple file program submission. It separates into the following two parts:

(1) Preprocessor: the beginning part of source statement or preprocessing statement #include<[library]> which can be parsed by using ASTPreprocessorStatement. The user can use several external files to support main files for solving one programming problem. The filename of the sources should be the same in file upload process because it will be linked via parameter search. Custom libraries by the user can be written in this defined statement #include "[external_sources_filename]" as a rule. The double quote indicates its user custom libraries or external sources. The system will process linker-fusion process to combine linked multiple program file.

(2) Content: this part contains the body of source program or other parts below preprocessor statements.

4.2. Multifile Linker in Dynamic Grading. Dynamic grading use compilation for handling complex multifiles to be linked, as shown in Figure 2. As an example for linking complex multifile program C or C++ programming language, it uses the linking process of GCC compilation process. It uses two input: main source and path of external source (multiple files paths). The differentiation from static is the type of the main source in here which will become .bin files from ProcessBuilder. The linking process in the system is possible with GCC command by using the parameter in backend program as an example:

> gcc–o [main].exe [multiple_files_path] –w (for C complex multifile program) > g++ –o [main].exe [multiple_files_path]–w (for C++ complex multifile program)

Multiple_files_path expression is files' path of external sources used in main_source. The command will process the main program in the compilation process to become an object and call every external source in the linking process. The linking process will continue to analyze and expand the preprocessor to make every external source become intended object of executable file with –o command. The –w command used to give warning error feedback such as preprocessor is not linked in case of file not found which makes compilation process fail.

4.3. Multifile Linker in Static Grading. The process still has two inputs: main source (raw source) and the external source path of multiple files which are already generated in the preparation process. The two inputs will be linked and fused in the next process, as shown in Figure 3 by linker and fusion. Static grading using source program linker-fusion algorithm is in Algorithm 1. Final output is a fused program with all the source code as a whole. The fused source code will not be processed through compilation but processed with code analysis using AST to parse element of program in external source used in the main program.

The algorithm is divided into four steps:

Step 1: get preprocessor statement set *ps* of the main source program by Eclipse CDT/JDT API features which can generate AST of main source using getAllPreprocessorStatements ( ).

Step 2: handling preprocessor statement set *ps*. Every *ps* found will be filtered by using patternMatch( ) to detect

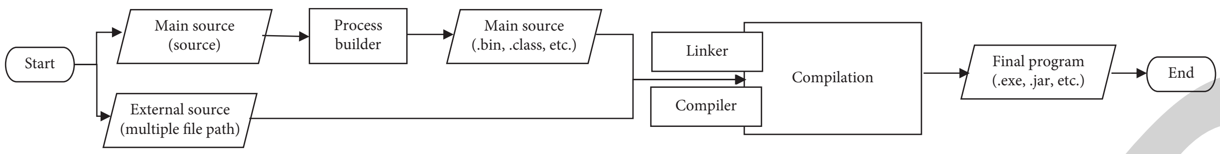

Start
Main source (source)
Process builder
Main source (.bin, .class, etc.)
External source (multiple file path)
Linker
Compiler
Compilation
Final program (.exe, .jar, etc.)
End

Figure 2: Complex multifile program linker in dynamic grading.

```
    **Variable**:
    ps[]: preprocessor statement pfname[]: preprocessor filename
    fpath: file path metadata in database fmp: final multipath string
    es: external source to be fused
    **Input**:
    ms: main source of the program
    mp[]: multipath in database by assignment id
    **Output:** final fused source fs
    **Begin**:
(1)  Step 1: Analyze preprocessor in ms using AST
(2)    if ms contains user defined custom library
(3)      add to ps[]
(4)  Step 2: Handle ps[] in ms to get pfname[]
(5)    if ps[] size > 0
(6)      for i = 1 to ps[] size
(7)        *regex ⟵ "¥"([^¥"]*)¥""
(8)          while patternMatch (regex) = = true
(9)            add to pfname[i]
(10) Step 3: Link external sources using mp[] to get pfname[]
(11)   for i = 1 to mp[] size
(12)     for j = 1 to pfname[] size
(13)       if (mp[i] contains pfname[j]) fpath ⟵ mp[i]
(14)       if (j = = 1) fmp ⟵ fmp concat (mp[i])
(15)       else fmp ⟵ fmp concat (mp[i]) concat (,)
(16) Step 4: Fusion es using fmp to fs
(17)   fs ⟵ ms
(18)   if fmp length > 0
(19)     for i = 1 to fmp length
(20)       ∈es ⟵ searchContent (fmp(∈ms))
(21)       fs ⟵ fs concat (∈es)
(22) End
```

Algorithm 1: Multifile program linker and fusion algorithm.

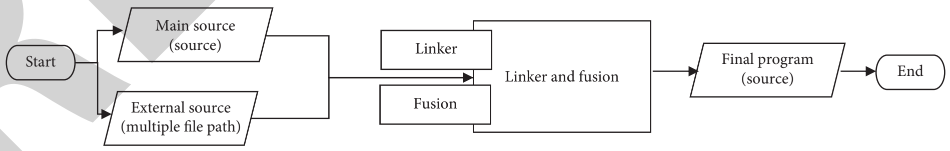


Figure 3: Multifiles linker in static grading.

user custom libraries defined by the following format:# [TYPE_OF_LANGUAGE_IMPORT_WAY(import/include)]<space>"[FILENAME]." Regex is used to parse preprocessor filenames into pfname set.

Step 3: linking process by analyzing the file path database *mp*. Each *pfname* found *mp* will generate fmp as final linked string query for every external source used in main source of the program.

Step 4: final step of the process is to get fused source code *fs*. The searchContent() function will analyze the code by using textual search (e.g., .dll,.h) or AST (source code which can be parsed) to get the contents of the files, denoted by ∈ *es*, used by the main source of the program. These contents are fused into one source file.

4.4. Dynamic Grading. The dynamic grading method covers multiprogram file submission for solving one programming assignment. The main source file triggered in here stated as a student program with several preprocessor statements is used to direct the external sources files. The first phase is

```
     Variable:
     ASTdata[]: program AST data map
     fASTdata[]: final refactored program AST data map
     Input:
     pAST: program AST
     sb[]: syntax bank
     Output: final standardized program AST fsAST
     Begin:
(1)  Step 1: Traverse and visit pAST with ASTVisitor class, do Step 2
(2)  Step 2: Handle AST node for AST data mapper to get ASTdata[]
(3)    Index = 0
(4)    for every AST node
(5)    initialization ASTdata[] key (see detail in Table 3)
(6)    node = AST node, current tag = AST tag
(7)    if current tag = function statement or expression
(8)      syntax = nodeParser(node)
(9)      if syntax not exists on sb[]//syntax classifier
(10)       inner user function node
(11)     else
(12)       inner standard function node
(13)   if current tag ≠ previous tag
(14)     exit no = index (exit AST node branch tree)
(15)   node info = flag based on node and tag information, process no = index
(16)   index ++
(17)   add ASTdata[index]
(18) Step 3: Handle AST data for refactoring to get final ASTdata[]
(19)   fASTdata[] = RefactoringCore(ASTdata[i])
(20) Step 4: Rebuild final source from fASTdata[] to get fsAST
(21)   for i = 0 to fASTdata[] size
(22)     fsAST = fsAST add node(fASTdata[i])
(23) End
```

Algorithm 2: Program standardization using AST algorithm.

Table 3: AST data map.

Key	Description
Index	Used for indexing of visiting and traversing process sequences while using *ASTVisitor*
Node	Visited syntax node
Parent node	Node parent to indicate its parent
Child node	Node child to indicate its child by node parser
Flag	Indicating node type (i). Root (node does not have parent and has child or its entry, and it also can indicate starting of program function) (ii). Parent (node has parent and child) (iii). Child (node has parent only, or its leaf node)
Syntax classifier	Indicating syntax to distinguish whether its user defined syntax or standard programming language syntax; it can be distinguished by comparing standard programming language syntax data provided and learned in database
Process	Indicating process entry (starting process) and exit number of node

program preparation, involving program builder process to prepare .bin file for C/C++ and class for java or other languages will be supported in future update. It also processes external source locator for linking process based on the preprocessor statement. Linking and compilation task example for C and C++ program use GNU GCC compiler which produce executable file.exe of linked and final program. Next process is using the executable file to run in DGRADER host machine with input from test case data. The running process of each test case produces an output.txt file. It will be compared with the test case expected output in database grading criterion. The grading process will be taken after achieving the comparison value of real output and expected output. Finally, final dynamic grading score will be produced.

4.5. Static Grading. The inputs of the static grading process include a student program with external sources (if any) and model programs. A student program and its external source will be finalized with program linker-fusion which was already explained before. In this case, both fused student

```
     Input:
     A: Standardized student program AST
     B: Standardized model program AST i
     Output: Count of similarity node
     Begin:
(1)  Step 1: GenerateMap(A, B)
(2)    A ⟶ set info tree based on Map key for A tree
(3)    B ⟶ set info tree based on Map key for B tree
(4)    global TagMap ⟵ ∅
(5)    global GlobalNodeMap ⟵ ∅
(6)    for each root node $r \in A \cap B$
(7)      do { A⟵AST of r in A
              B⟵AST of r in B
              Similarity Matching (A, B) }
(8)  Step 2: SimilarityMatching(A, B)
(9)    ns, cp ⟵ 0
(10)   local LocalNodeMap ⟵ ∅
(11)   for each (ANode, BNode) ∈ (A, B)
(12)     if (ANode, BNode) = (ATag ANodeNo, BTag BNodeNo)
(13)       TagMap ⟵ TagMap ∪ {ATag ↔ BTag}
(14)       LocalNodeMap ⟵ LocalNodeMap ∪ {ANodeNo ↔ BNodeNo}
(15)     else if (ANode, BNode) = (AChildNo := $a$ op a', BChildNo: = $b$ op b')
(16)       SimilarityMatching(a, b)
(17)       SimilarityMatching(a', b')
(18)       if isLocal(a) and is Local (b) and TagMap(a, b) is equal
(19)         LocalNodeMap ⟵ LocalNodeMap ∪ {a ↔ b}
(20)         ns+1, cp+1
(21)       else if
(22)         GlobalNodeMap ⟵ GlobalNodeMap ∪ {a ↔ b}
(23)         cp+1
(24)       else if. . .
(25)       else break
(26) End
```

Algorithm 3: AST simple tree matching with tag classifier algorithm.

Table 4: Experiment setup for testing dynamic grading.

	Assignment name	Main goal	No. of test cases	Concurrent process
Task 1	Calculate numbers based on data	Calculate n of i_n data	10	7
Task 2	Reverse Fibonacci number	Print n reversed Fibonacci number	15	10

program and model programs will be parsed and produce their program AST. Next process will conduct refactoring of both programs. In our preferred way, model programs are already have been standardized before by the course assignment creator in assignment registration menu to save total time of analysis process. Only student program will be refactored as standardization rule in program matching. This standardization as "one-rule" for avoiding some variances issues will impact the accuracy result in the matching process. After this process, both sides can be compared using program matching using modified AST simple tree matching-pattern algorithm. The task of standardization and program matching will breakdown in Section 4.5.1.

4.5.1. Program Standardization. The program standardization algorithm is shown in Algorithm 2. It is an essence work to solve program matching issues such as code variance in programs.

It starts with program AST and continues with traversing and visiting tree process using ASTVisitor. In here, the approach introduces AST data mapper rather than using raw program tree or AST directly to manipulate program which is required in the refactoring process to ease the process of data representation and rebuilding program tree. The AST data mapper stores AST information from ASTVisitor by using map with indexing key (*ASTdata* key written in Table 3). After mapping process, it continues to refactoring process by calling RefactoringCore() to modify or transform the program by using this data map. This procedure will create final *fASTdata*. It will be used for rebuilding final standardized program source code and program tree representation in feedback.

Program refactoring consists of the following steps.

Step 1: removal of problematic code such as syntax error, expression error, and any other with tag problem in *ASTdata* by problem binder which processed while

Table 5: Dynamic grading result.

	Result	
	Grading feedback	Average execution time (ms)
Task 1	Yes	20
Task 2	Yes	136

Table 6: Static programming assignment example.

	Assignment name	Main goal
Task 1	Calculate numbers (data)	Calculate n of i_n data
Task 2	Reverse fibonacci number	Print n reversed fibonacci number
Task 3	Coin changing	Consider the problem of making change for n cents using the fewest number of coins
Task 4	*Bitonic* tour	Input: n Points: $v_1, ..., v_n$ on a plane with different x coordinates; d_{ij}: distance between any pair of points (v_i, v_j), $i \neq j$; output: a *bitonic* tour with smallest distance

traversing the program using AST and mapped into *ASTdata* mapper. The problem binder takes advantage of parser to detect syntactical problem. As an example for traversing the C and C++ program using Eclipse CDT *ASTVisitor CPPASTProblem.*

Step 2: identifying and removal of unused function in *ASTdata.* This task deletes its related elements by using start–end index of the node which is already mapped in *ASTdata.*

Step 3: reordering function calling sequences. In this case, main function will become first function call in sequence as the rule of program. The process continues to detect other call function by creating *ASTroot* data mapper. It only targets root type of the node with the root tag filter or classifier. The node is considered as a root node according to flag root detail in Table 3. The result of transformation can be seen from transformation from original AST into root AST.

Step 4: the final step is producing a final standardized program from *ASTroot.*

4.5.2. Program Matching. The program matching process requires two program trees as an input which are the student program and model program *i*. Both programs are already standardized automatically by the system in the previous process to achieve same structure as one-rule policy in process of program matching.

In program matching, AST simple tree matching is chosen to compare program AST which takes advantage of dynamic programming to calculate maximum node-pair between compared tree program with semantic similarity [29]. However, this algorithm has an issue with overhead memory as it does the code changes when code variance occurred in matching process to make the variance for both compared programs unified. It is also stated in the result test that some code changing activity fail in the process. This will impact the matching similarity accuracy score. Finally, we have improved the approach with AST simple tree matching algorithm, as shown in Algorithm 3.

The algorithm is modified to use advantage of AST tag classifier concept which can be implemented by the method in program AST traversing with any parser. This will solve the overhead and the variance issues.

The improved program matching algorithm has two steps:

Step 1: both programs will be mapped into *ASTMap* which contains key information. It uses root node *r* which is already ordered by following the rule in the standardization process. Tag and node are used as matching parameters. *TagMap* is used to take advantage of the AST tag classifier in tag matching comparison. *GlobalNodeMap* and *LocalNodeMap* are used for node element which will be considered equal if encountered in the same tag in the same root node position.

Step 2: after both program trees are mapped into *ASTMap*, the process will invoke SimilarityMatching( ). Every index data in map will be compared and paired. Accumulate the global *TagMap* and compare *GlobalNodelMap* with *LocalNodeMap* per indexed data in *ASTMap.* Next process is to encounter the exact tag to be compared with it pairs and add into *TagMap* in tag matching task. *LocalNodeMap* used for detection of the identical node in node matching. As stated before, as long as the node follows the same tag and position it will be considered as equal otherwise it will add pattern to find the matching target invoking recursive SimilarityMatching( ) call. For every matching node with same position in local A and B will increase node similarity and pattern count. Total accumulate *ns* will be compared to maximum node coverage of both programs. Finally, the final score can be calculated based on how many maximum nodes are covered by *ns* in the grading process.

5. Experimental Analysis

5.1. Dynamic Grading. In dynamic grading, testing conduct with 2 tasks in Table 4. The concurrent process is used to measure system capability to handle concurrent grading process when students finalize their submission at the same time.

Table 7: General testing solution program condition.

	LOC	Function	Max node	Unused node	Standardization	
					Transform node	Program efficiency improvement rate (%)
Solution 1	12	1	40	6	38	1.9
Solution 2	48	7	94	43	49	52.128
Solution 3	55	9	182	24	158	13.19
Solution 4	120	14	424	35	389	8.25

Table 8: General testing condition.

	Task 1	Task 2	Task 3	Task 4
No. of model program	2	2	5	8
Concurrent process	4	3	7	4

Table 9: General static grading result.

	Result		
	Grading feedback	Analysis time (ms)	Expected result
Task 1	Yes	88	Yes
Task 2	Yes	90	Yes
Task 3	Yes	1022	Yes
Task 4	Yes	678	Yes

All submission results shown in Table 5 produce grading feedback both successful or error result explanation with the score. The feedback shows success or failure with real and expected output of each test case. In here, testing activity also considers average execution time which measures the time needed by the system to finish the job. Finally, it is successful to achieve dynamic grading mechanism. The concurrent process also can be handled by automatic queuing process and multithreading.

5.2. Static Grading. In static grading, programming assignment tasks are shown in Table 6. Task 1 and task 2 are reused in this testing as DGRADER offer flexibility grading assessment method change by updating the assignment configuration. The lecturer just needs to add model programs in the assignment material. Back to testing focus, each sample solution has its behaviour and condition according to Table 7 such as Line of Code (LOC), number of functions, max number of nodes (generated by traversing program AST), and number of unused nodes (code node which are not used in the program).

Standardization data present transforms result from maximum number of nodes to number of transform nodes as automatic transform by program standardization in the system. The number of transform nodes implies the efficiency of the standardized program. The program efficiency improvement rate can be calculated by comparing original maximum number of nodes and number of transform nodes. For example, solution 2 data showed significant 52.128% program efficiency improvement.

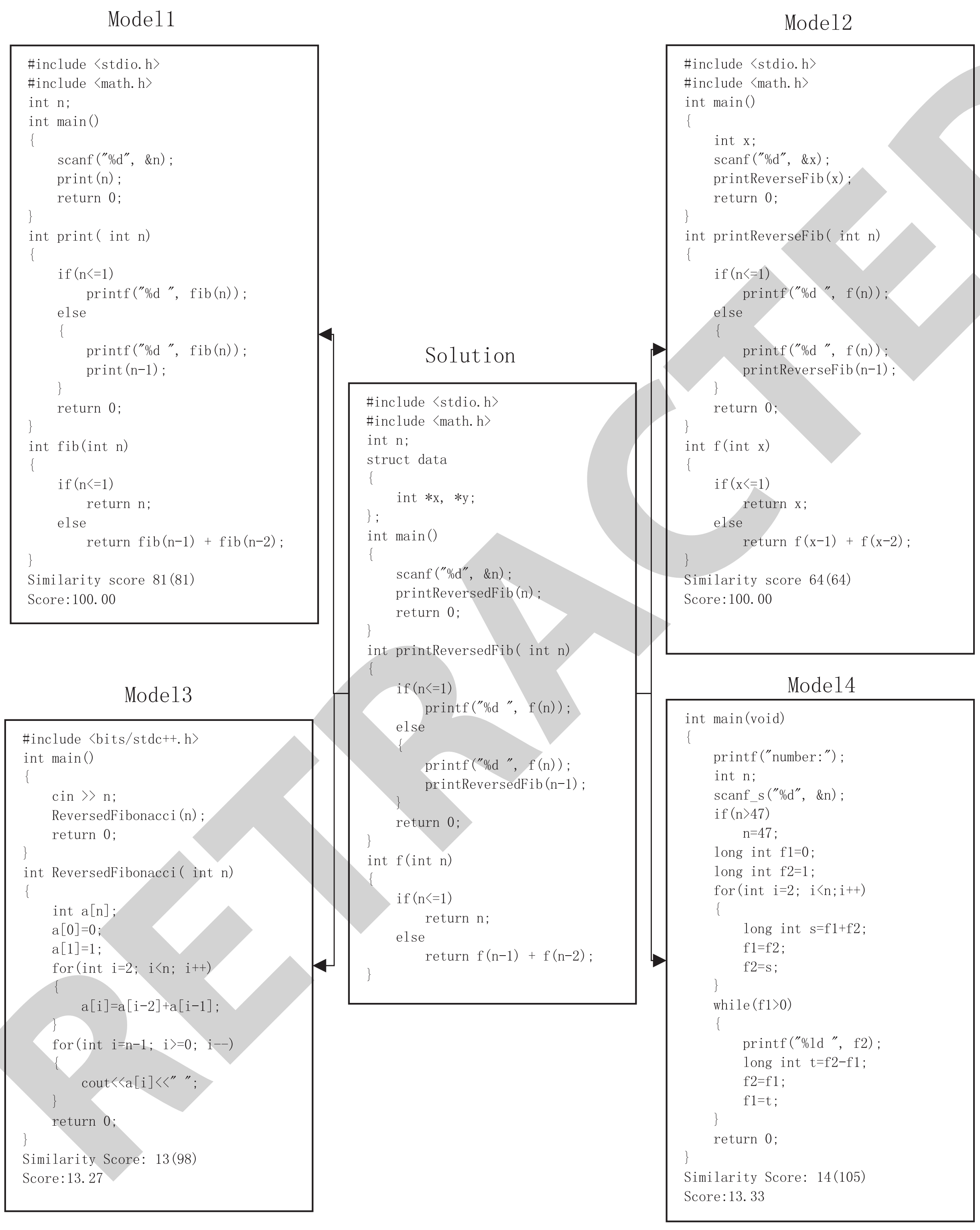
Based on Table 8 as the testing condition, the result is shown in Table 9. All solution submitted successfully gives grading feedbacks and results. Expected result means that every variance tolerant achieve full 100% mark as it uses tag from the AST classifier. It makes every variation occurred in the program solved or tolerated as long as it has true condition from tag and node matching in program matching. The expectation of testing by manual assessment produced. Finally, static grading approach implemented in DGRADER can be found reliable as it is giving good precision of grading score and is also proofed by manual assessment.

5.3. Reliability and Grading Precision Testing. This testing is focused in reliability of the final grading score in the static grading assessment. In here, assignment "reverse Fibonacci number" in the previous assignment sample is chosen. The approach uses 1 solution submission and sees its comparison with 4 models (chosen) provided in the assignment. Solution and model programs are shown in Figure 4. The solution program is chosen solution 2 in the previous test, and its condition can be seen in Table 7, and model condition is listed in Table 10.

In here, testing activity hypothesis for model 1 and model 2 will give perfect grading score in the final result. Both the models are nearly same to program solution and also its algorithm which used recursive function to do the job for printing reversed Fibonacci number. The purpose is to check the reliability and precision of logically similar programs by manual assessment. Model 3 is actually .cpp program which uses array approach, and model 4 uses more variables to save values for producing reversed Fibonacci number without recursive function call. They adopt different algorithms to the student solution.

It is clearly evident from Table 11 that similar nodes of solution to models 1 and 2 have 100% final coverage. As stated before, model 1 and model 2 are not exactly the same to the solution. The perfect coverage achieved by variance tolerance successfully tolerates the code variance issues in program matching process. Variance type divided into 3 categories is used based on 9 listed code variance issues in Table 1. In here, the division is based on their related impact into the issue listed variable (name), function (invocation or call, expression, naming), and control structure (compound statement, redundancy, structure order, code format, and algorithm).

Starting analysis with model 1 which is nearly the same, but program solution has code structure data {int x^*, int y^*}. In matching, the pattern increased as it tries to find the node first related to the code with ending in not finding any possible node related to that code. In here, matching still processed and ignored as tolerant in the control structure as the code categorized to impact in the compound statement



Model1

```
#include <stdio.h>
#include <math.h>
int n;
int main()
{
    scanf("%d", &n);
    print(n);
    return 0;
}
int print( int n)
{
    if(n<=1)
        printf("%d ", fib(n));
    else
    {
        printf("%d ", fib(n));
        print(n-1);
    }
    return 0;
}
int fib(int n)
{
    if(n<=1)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
Similarity score 81(81)
Score:100.00
```

Model2

```
#include <stdio.h>
#include <math.h>
int main()
{
    int x;
    scanf("%d", &x);
    printReverseFib(x);
    return 0;
}
int printReverseFib( int n)
{
    if(n<=1)
        printf("%d ", f(n));
    else
    {
        printf("%d ", f(n));
        printReverseFib(n-1);
    }
    return 0;
}
int f(int x)
{
    if(x<=1)
        return x;
    else
        return f(x-1) + f(x-2);
}
Similarity score 64(64)
Score:100.00
```

Solution

```
#include <stdio.h>
#include <math.h>
int n;
struct data
{
    int *x, *y;
};
int main()
{
    scanf("%d", &n);
    printReversedFib(n);
    return 0;
}
int printReversedFib( int n)
{
    if(n<=1)
        printf("%d ", f(n));
    else
    {
        printf("%d ", f(n));
        printReversedFib(n-1);
    }
    return 0;
}
int f(int n)
{
    if(n<=1)
        return n;
    else
        return f(n-1) + f(n-2);
}
```

Model3

```
#include <bits/stdc++.h>
int main()
{
    cin >> n;
    ReversedFibonacci(n);
    return 0;
}
int ReversedFibonacci( int n)
{
    int a[n];
    a[0]=0;
    a[1]=1;
    for(int i=2; i<n; i++)
    {
        a[i]=a[i-2]+a[i-1];
    }
    for(int i=n-1; i>=0; i--)
    {
        cout<<a[i]<<" ";
    }
    return 0;
}
Similarity Score: 13(98)
Score:13.27
```

Model4

```
int main(void)
{
    printf("number:");
    int n;
    scanf_s("%d", &n);
    if(n>47)
        n=47;
    long int f1=0;
    long int f2=1;
    for(int i=2; i<n;i++)
    {
        long int s=f1+f2;
        f1=f2;
        f2=s;
    }
    while(f1>0)
    {
        printf("%ld ", f2);
        long int t=f2-f1;
        f2=f1;
        f1=t;
    }
    return 0;
}
Similarity Score: 14(105)
Score:13.33
```

Figure 4: Feedback result of program matching trace analysis.

and structure order. It also occurs function naming variance print() and fib() in model compared with printReversedFib() and f() in solution which are successfully tolerated as it is expected to be considered the same. Left remaining codes are exactly the same with solution, which means similarity node coverage of program solution and model 1 is perfect. It covers 81 nodes of model 1 program with 100% (tolerate all variances) in final coverage.

In model 2, there exists variable variances of node x which is node n in program solution. In matching process, it is found that x corresponds to node n for the following code. The matching process considered in this node is the same as

Table 10: Model program condition.

	LOC	Function	Max node	Variance		
				Variable	Function	Control structure
Model 1	48	3	81	2	2	1
Model 2	48	3	64	1	2	1
Model 3	25	2	98	1	1	2
Model 4	22	1	105	4	0	2

Table 11: Static grading precision result.

	Similarity node	Max node	Final coverage (%)	Variance tolerance			Total pattern matching process
				Variable (%)	Function (%)	Control - Structure (%)	
Model 1	81	81	100	—	100	100	394
Model 2	64	64	100	100	100	100	387
Model 3	13	98	13, 26	100	100	<10	137
Model 4	14	105	13, 33	100	100	<20	159
Total pattern							1077
Precision matching							>98

Table 12: Existing similar program comparison (main features).

Approaches	Main features					
	Supported languages	Complex multifiles analysis	Platform	Work mode	Grading metric	
					Dynamic	Static
Cellidh	Java, C++	No	Web	Standalone, competitive learning	Code correctness	Semantic error detection, verification
ASSYST	C/C++	No	Web	Standalone, competitive learning	Code correctness, run time check	Code analysis, complexity matrices
Marmoset	Multilanguage	No	Web	Standalone	Code correctness	Code analysis (model)
Web-CAT	Multilanguage	No	Web (Java)	Standalone, plugins	Code correctness, completeness	Validity check (model)
eGrader	Java	No	Desktop (Java)	Standalone	Code correctness	Structure matching (model)
AutoLEP	C/C++	No	Desktop (C#)	Standalone	Code correctness	Similarity matching (model)
Quimera	C/C++	No	Web	Standalone, competitive learning	Code correctness	Code quality and accuracy (model)
DGRADER	C/C++, Java	Yes	Web (Java)	Standalone, API services, competitive learning	Code correctness, run time check	AST similarity matching (model)

expected, as algorithm uses the AST tag classifier in nodes. Besides, structure variance occurs as declaration of x inside main() compared to program solution which is global variable. The matching process is also successful to tolerate this variance and gives similarity 64 node coverages perfectly as expected.

However, models 3 and 4 as expected have low final coverage because both are completely different. The major factor is because both models have distinguished control structure such as algorithm, code format, and related factor which is categorized in this category. Model 3 as stated before is .cpp but the system still is capable to compare .c program solution with this model which is a plus point. Models 3 and 4 use different approach from solution that uses recursive method. It impacts the matching process which gives less node coverage and tolerant percentage. The issues can be solved by providing more template program to the system for assignment problem. Finally, final grading score still gives correct score as expected.

5.4. Case Study Similar Program Comparison Analysis. The columns in Table 12 refer main features of this objective comparison parameter. From the above parameter, we compare DGRADER as our web-based automatic grading system with the other similar programs which provide flexible grading analysis.

The first key element for comparison analysis is programming language support. Marmoset and Web-CAT are strong in this aspect as they support multilanguages.

Table 13: Existing similar program comparison (scoring features).

Approaches	Scoring features					
	Categorical grading	Numerical grading	Feedback		Ranking	Plagiarism detection
			Error	Result analysis		
Cellidh	No	No	Yes	Yes	Yes	Yes
ASSYST	Yes	No	Yes	Yes	Yes	Yes
Marmoset	No	No	No	Yes	No	No
Web-CAT	No	No	Detailed (highlighting)	Code style (highlighting)	No	No
eGrader	No	Yes	Yes	Program structure	No	No
AutoLEP	No	Yes	Detailed (report)	Semantic analysis	No	No
Quimera	No	Yes	Yes	Yes	No	No
DGRADER	Yes	Yes	Detailed (report)	Standardization, AST map, transform graph, function call, AST visualization, and matching trace analysis (semantic analysis)	Yes	Configurable

The second one is complex multifile program analysis feature. Only DGRADER provides complex multifile program analysis for the user to solve one programming problem within more than one modules, libraries, or files. It improves flexibility to solve complex problems with an efficient approach without rewriting program if some libraries exist.

In platform perspective, web platform has become popular consideration rather than becoming a local tool such as AutoLEP and eGrader. Nowadays, everything can be accessed online easily by using browser or mobile device which increases the portability which adds value for flexibility scale.

Next on aspect of work mode, common tools only work as standalone or specific usage. Web-CAT provides plugin for their integration with other platform which require installation. Only DGRADER can work as standalone or integrated with other system by providing its API services which potential for widespread usage. It also delivers with MOOCs with user course management and programming contest platform as feature to support competitive learning. Cellidh, ASSYST, and Quimera also support this competitive learning. It is purposed to increase interest of the user as the content in the system can become user content based.

In grading metrics, compared tools provide flexible grading analysis, which are dynamic and static. However, each approach is different as listed. In dynamic approach, it can be generalized that all approaches are tested with code correctness by using test case input and the output is compared with the expected output. In static, there are various approaches using code analysis. DGRADER uses AST similarity matching with model programs. The approach with model is popularly used by the existing tools. This model is more practical and easier to be measured but requires more models to increase the accuracy and grading precision.

The columns in Table 13 refer the scoring features of this objective comparison parameter. DGRADER comes with various result analyses as their instant feedback and scoring. Instant feedback includes error explanation if any error occurred specially in dynamic grading. In static grading assessment, present feedback such as linked-fused program source, standardized program source, and mapped into AST data map can be read by the user in result analysis. It also presents transformed graph structure to see change by automatic standardization process, function call sequence, AST visualization of program, and matching trace analysis.

Final scoring provided both in number range 0–100 and categorical score with stars to make the system more interactive. User ranking also provided to stimulate usage and improve user programming and problem solving skill through the reward system with point and level. Plagiarism detection feature is also considered in this scoring feature as it will impact the final grading score. In these compared tools, only Cellidh, ASSYST, and our tool DGRADER provide this feature. The programming assignment creator also can enable plagiarism detection features.

Finally, all comparison analysis can measure to see improvement of each compared tools. Back to DGRADER, our tool currently limited programming language as listed. However, API services will be potential for future development of cross platform integration which means more support of programming language and other features. DGRADER also provides meaningful and informative feedback with novelty complex multifile program analysis feature.

6. Conclusions

In this paper, we have presented a novel technique to handle complex multifiles program with flexible static and dynamic grading. It is implemented practically as an automatic grading system platform called DGRADER. In order to deal with multifiles program, the dynamic analysis process takes an advantage of the compiler in linking process to compile complex multifile program. The static analysis process uses the presented complex multifiles program linker-fusion algorithm which parsed preprocessor from program AST to find other external sources. It is successful to link and fuse elements in external sources which are used in the main program. In static analysis, code variance issues in program matching are tackled by the improved algorithm of AST simple tree matching. The AST tag classifier creates tolerant factor of variances from the compared node of solution and model programs. Program standardization also contributes to transform the program by following the rules in program matching. The results have shown good accuracy in final grading precision as expected in the case of sufficient model programs.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant nos. 61977020 and 61672191), Natural Science Foundation of Heilongjiang Province (Grant no. LH2019F046), Harbin Science and Technology Innovation Talents Research Project (Grant no. 2016RAQXJ013), and Doctoral research fund of Harbin University (Grant no. HUDF2019101).

References

[1] S. Li, X. Xiao, B. Basset, T. Xie, and N. Tillman, "Measuring code behavioral similarity for programming and software engineering education," in *Proceedings of the ACM 38th IEEE International Conference on Software Engineering Companion*, pp. 501–510, Austin, TX, USA, May 2016.

[2] S. Gulwani, I. Radiček, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*, pp. 465–480, New York, NY, USA, June 2018.

[3] T. Wang, J. Xu, X. Su, C. Li, and Y. Chi, "Automatic debugging of operator errors based on efficient mutation analysis," *Multimedia Tools and Applications*, vol. 78, no. 21, pp. 29881–29898, 2019.

[4] D. G. Kay, T. Scoot, P. Isaacson, and K. A. Reek, "Automated grading assistance for student program," *ACM SIGCSE Bulletin*, vol. 26, no. 1, pp. 381-382, 1994.

[5] P. Li and L. Toderick, "An automatic grading and feedback system for e-learning in information technology education," in *Proceedings of the ASSE Annual Conference and Exposition for Emerging Computing and Information Technologies*, pp. 1–11, Seattle, Washington, June 2015.

[6] C. Wilcox, "Testing strategies for the automated grading of student program," in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education—SIGCSE '16*, pp. 437–442, Memphis, TN, USA, March 2016.

[7] J. Qi, G. Jiang, G. Li, Y. Sun, and B. Tao, "Intelligent human-computer interaction based on surface EMG gesture recognition," *IEEE Access*, vol. 7, pp. 61378–61387, 2019.

[8] G. Li, L. Zhang, Y. Sun, and J. Kong, "Towards the sEMG hand: internet of things sensors and haptic feedback application," *Multimedia Tools and Applications*, vol. 78, no. 21, pp. 29765–29782, 2019.

[9] G. Conole and B. Warburton, "A review of computer-assisted assessment," *ALT-J*, vol. 13, no. 1, pp. 17–31, 2005.

[10] K. M. Ala-Mutka, "A Survey of automated assessment approaches for programming assignments," *Computer Science Education*, vol. 15, no. 2, pp. 83–102, 2005.

[11] P. Ilhantola, T. Ahoniemi, V. Karavirta, and O. Seppala, "Review of recent systems for automatic assessment of programming assessment," in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research—Koli Calling '10*, pp. 86–93, Koli, Finland, October 2010.

[12] T. Wang, X. Su, and P. Ma, "Program normalization for removing code variations," in *Proceedings of the 2008 IEEE International Conference on Computer Science and Software Engineering*, pp. 306–309, Hubei, China, December 2008.

[13] T. Wang, X. Su, and P. Ma, "Function inlining algorithm for program analysis," in *Proceedings of the 2009 IEEE International Conference on Computational Intelligence and Software Engineering*, pp. 1–4, Wuhan, China, December 2009.

[14] S. M. Arifi, A. Zahi, and R. Benabbou, "Semantic similarity based evaluation for C programs through the use of symbolic execution," in *Proceedings of the 2016 IEEE Global Engineering Education Conference*, pp. 826–833, Abu Dhabi, UAE, April 2016.

[15] A. N. Jacobvitz, A. D. Hilton, and D. J. Sorin, "Multi-program benchmark definition," in *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 72–82, Philadelphia, PA, USA, March 2015.

[16] K. Matthews, T. Janicki, L. He, and L. Patterson, "Implementation of an automatic grading system with an adaptive learning component to affect student feedback and response time," *Journal of Information System Education*, vol. 23, no. 1, pp. 71–83, 2012.

[17] M. Pozenel, L. Furst, and V. Mahnic, "Introduction of the automated assessment of homework assignments in a university-level programming course," in *Proceedings of the 2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 761–766, Opatija, Croatia, May 2015.

[18] T. Wang, X. H. Su, and P. J. Ma, "Semantic similarity-based grading of student programs," *Information Software Technology*, vol. 49, no. 2, pp. 17–31, 2007.

[19] D. Fonte, D. Cruz, A. L. Gancarski, and P. R. Henriques, "A flexible dynamic system for automatic grading of programming exercises," in *Proceedings of the 2nd Symposium on Language, Applications and Technologies*, pp. 129–144, Dagstuhl, Germany, 2013.

[20] S. D. Benford, E. K. Burke, E. Foxley, and C. A. Higgins, "The Ceilidh system for the automatic grading of students on programming courses," in *Proceedings of the 33rd Annual on Southeast Regional Conference ACM-SE 33*, Clemson, SC, USA, March 1995.

[21] D. Jackson and M. Usher, "Grading student programs using ASSYST," in *Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, pp. 335–339, San Jose, CA, USA, February 1997.

[22] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez, "Experiences with marmoset," *ACM SIGCSE Bulletin*, vol. 38, no. 3, pp. 13–17, 2006.

[23] S. H. Edwards and M. A. Perez-Quinones, "Web-CAT," *ACM SIGCSE Bulletin*, vol. 40, no. 3, p. 328, 2008.

[24] F. Alshamsi and A. Elnagar, "An automated assessment and reporting tool for introductory Java programs," in *Proceedings of the 2011 International Conference on Innovations in*

Information Technology (IIT), pp. 324–329, Abu Dhabi, UAE, April 2011.

[25] T. Wang, X. Su, P. Ma, Y. Wang, and K. Wang, "Ability-training-oriented automated assessment in introductory programming course," *Computers & Education*, vol. 56, no. 1, pp. 220–226, 2011.

[26] D. Fonte, I. V. Boas, D. Cruz, A. L. Gancarski, and P. R. Henriques, "Program analysis and evaluation using quimera," in *Proceedings of ICEIS*, pp. 209–219, Wroclaw, Poland, June 2012.

[27] T.-H. Wang, "Web-based dynamic assessment: taking assessment as teaching and learning strategy for improving students' e-Learning effectiveness," *Computers & Education*, vol. 54, no. 4, pp. 1157–1166, 2010.

[28] D. Muñoz de la Peña, F. Gómez-Estern, and S. Dormido, "A new internet tool for automatic evaluation in control systems and programming," *Computers & Education*, vol. 59, no. 2, pp. 535–550, 2012.

[29] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," in *Proceedings of the 2005 International workshop on Mining Software Repositories*, pp. 1–5, Saint Louis, MO, USA, May 2005.