Automated Assessment in Computer Science Education: A State-of-the-Art Review

JOSÉ CARLOS PAIVA, JOSÉ PAULO LEAL, and ÁLVARO FIGUEIRA, CRACS-INESC TEC and DCC-FCUP, Portugal

Practical programming competencies are critical to the success in computer science (CS) education and go-to-market of fresh graduates. Acquiring the required level of skills is a long journey of discovery, trial and error, and optimization seeking through a broad range of programming activities that learners must perform themselves. It is not reasonable to consider that teachers could evaluate all attempts that the average learner should develop multiplied by the number of students enrolled in a course, much less in a timely, deep, and fair fashion. Unsurprisingly, exploring the formal structure of programs to automate the assessment of certain features has long been a hot topic among CS education practitioners. Assessing a program is considerably more complex than asserting its functional correctness, as the proliferation of tools and techniques in the literature over the past decades indicates. Program efficiency, behavior, and readability, among many other features, assessed either statically or dynamically, are now also relevant for automatic evaluation. The outcome of an evaluation evolved from the primordial Boolean values to information about errors and tips on how to advance, possibly taking into account similar solutions. This work surveys the state of the art in the automated assessment of CS assignments, focusing on the supported types of exercises, security measures adopted, testing techniques used, type of feedback produced, and the information they offer the teacher to understand and optimize learning. A new era of automated assessment, capitalizing on static analysis techniques and containerization, has been identified. Furthermore, this review presents several other findings from the conducted review, discusses the current challenges of the field, and proposes some future research directions.

CCS Concepts: • **Applied computing → Computer-assisted instruction**; **Computer-managed instruction**; **Interactive learning environments**; *E-learning;*

Additional Key Words and Phrases: Automated assessment, computer science, programming, feedback, learning analytics

ACM Reference format:
José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Trans. Comput. Educ.* 22, 3, Article 34 (June 2022), 40 pages.
https://doi.org/10.1145/3513140

J. C. Paiva was supported by Ph.D. grant 2020.04430. BD from FCT - Fundação para a Ciencia e a Tecnologia (Portuguese Foundation for Science and Technology), Portugal.

Authors' address: J. C. Paiva, J. P. Leal, and Á. Figueira, CRACS-INESC TEC and DCC-FCUP, Rua do Campo Alegre, 817, Porto, Portugal; emails: up201200272@fc.up.pt, zp@dcc.fc.up.pt, arfiguei@fc.up.pt.

1946-6226/2022/06-ART34 $15.00
https://doi.org/10.1145/3513140

1 INTRODUCTION

Assessment involves the evaluation, measurement, and reporting of the academic readiness, learning progress, skill acquisition, and educational gaps of students. It constitutes an essential part of learning, as it provides a two-way feedback channel. On the one hand, it guides students and keeps them aware of whether they are achieving learning goals. On the other hand, it provides the teacher with information about the learning process, from the level of a complete course down to a single student in a specific topic. Furthermore, for many students, the amount and direction of their efforts negatively correlates with the time left for the next assessment, and positively with what is assessed, respectively [35, 128]. Hence, the assessment frequency and the quality of feedback it delivers are the principal drivers of academic achievement.

Providing manual assessment in educational contexts, where there is not a one-to-one relationship between the teacher and the learner, is rather challenging. Doing so in courses demanding regular and intense practice with tasks having infinite pathways to correct (and not so correct but acceptable) solutions, such as computer programming, becomes rather unfeasible. However, automated assessment can help to mitigate these issues, allowing instant and rigorous feedback on students' attempts without overloading the instructor. This enables learners to get enough practice with at least some indication of the correctness of their solutions, freeing instructors to focus on less repetitive tasks.

Automated assessment tools for **computer science (CS)** education have been there nearly for as long as learners started being asked to develop software [107]. Currently, there is a wide variety of such tools, and their value is well spread and accepted among practitioners. Research on this topic, however, is far from reaching the peak, as a simple search for the string "automated assessment programming" within the ACM Digital Library has returned more than 220,000 results for the past 10 years, with an increasing curve over the years. Many of these publications refer to tools created for the specific needs of a course, either because instructors fail to identify or integrate an existing suitable solution (e.g., difficulty in producing supported exercises or unavailability of the tool). Nevertheless, it is noticeable that there is a growing interest in exploring automated assessment of program features other than functionality, such as quality, behavior, readability, and security, as well as novel assessment methods to improve feedback.

Numerous studies surveying the literature on automatic approaches for assessing programming assignments exist [6, 68, 112, 204, 220]. However, part of them are already very outdated [6, 68, 112, 204], whereas the rest focus on comparing the characteristics of the tools [220] rather than the various facets of automated assessment tools, exploring the methods and techniques for each of them (except for Keuning et al. [130], who investigated several aspects of the feedback provided). Moreover, there is no review on these tools, to the best of the authors' knowledge, that examines them concerning learning analytics. In this article, we survey the state of the art on the automated assessment of CS assignments, focusing on five facets: supported exercise domains, testing techniques utilized, security measures adopted, feedback produced, and the information they offer the instructor to understand and optimize learning. A brief overview of tools providing automated assessment is also presented.

The rest of the article is organized as follows. Section 2 presents an overview of the history of automated assessment from the earlier systems until the state of the art in the 2010s. Section 3 presents the most relevant past surveys on automated assessment. Section 4 introduces the research questions and the methodology used to conduct this review. Section 5 describes the state-of-the-art review carried out, creating a panoramic view on supported exercise domains, methods used in the assessment, feedback provided to learners, and provided learning reporting and analytics. Finally, Section 6 discusses the results and presents the threats to the validity and main conclusions of this review, advocates some future research directions, and indicates our expectations related to future trends in automatic assessment of CS assignments.

2 BACKGROUND

Tools and techniques to assist instructors in assessing CS assignments, particularly those involving programming, have been designed and built for more than 60 years. In the earliest known automated assessment system [107], students "submitted" their assembly programs on punched cards, which were then run along with the grader program and a deck of cards with exercise data. If the program terminates normally, the grader punches "problem complete"; otherwise, "wrong answer" is punched into the card (it can also punch the cause of the error, but it requires manual entry as the computer stops). Such a simple system, with several limitations and requiring machine operators, made it possible to more than double the class size, without degrading the learning outcome, and even teaching programming by mail [107].

Following the advances in the field of programming, automated assessment systems have also evolved. First, new grader systems emerged that could grade programs written in a programming language rather than machine language [31, 79, 161]. These systems use a grader program that (1) injects learners' code into the grader as procedures, (2) calls these procedures with test data (also part of the program) one at a time, (3) optionally keeps track of the running time, and (4) marks tests as accepted or wrong by comparing results of a learner's procedure with the results of the solution. The evident limitations of this approach include the need for recompilation of the grader program for each specific problem, test data, and the set of learners' procedures, as well as the complexity of developing a grader (a plausible explanation to why all consider the same problem: finding the root of a given function to a given accuracy in a prescribed interval).

With a new and more versatile programming language (PL/1), Temperly and Smith [234] introduced in 1968 a method that called the grader as a precompiled library subroutine, eliminating the need to recompile the grader program. As this greatly simplified grading programs, learners were allowed to test their attempts against a subset of test data before submitting the final program. All runs were recorded along with the student's identification and execution time, which helped to limit the attempts they could perform. With even more relevance were the contributes of Hext and Winings [103] in the following year, who modified the operating system and developed a series of language processors to be able to run submitted programs in any programming language as batch-processed jobs. The program assessment consisted of comparing the stored test with the one obtained by executing the submitted program, storing a detailed report of the test results for later consultation.

As programming languages became more humanized and software more manageable during the 1980s, the complexity of automated assessment systems also decreased from complex and difficult to maintain systems to command-line scripts many times part of an installable tool [116]. For instance, Isaacson and Scott [116] present a command-line utility that takes as input a directory with one folder per student (where they place the program) and a predefined set of input/output test files to access, compile, and execute student programs, reporting the results by email. Interestingly, the work of Isaacson and Scott also addresses a known issue of this kind of system—security, previously "guaranteed" manually, relying on users and permissions of the operative system to deny access of the students to other directories. Running the submitted program as an instructor, however, exposes the instructor's account to a malicious student's program (e.g., an instruction to wipe the disk), as noticed by Reek [198]. Having that in mind, TRY [198] arranges for a wrapping directory to appear to the program as the root of the file system, blocking access to directories outside. In addition to that, the tool makes significant contributions in the students' perspective, enabling them to perform a few tests before sending the final program and alleviating the traditional character-by-character output comparison with a white space cleaning script.

At the same time, the static features of the source code, particularly readability and complexity, started catching research attention as important evaluation metrics beyond correctness and execution time [199, 240]. Nevertheless, the following decade was marked by the proliferation of automated assessment tools, on the line of the previous ones. Kassandra [245] was the first of its kind to support programming languages designed for scientific computing, also innovating in how it provides isolation through socket connections to exchange test data during the evaluation. ASSYST [118] incorporates additional metrics in the assessment, such as the amount of CPU time, code complexity, and code style, while allowing instructors to define weights for each test (and aspect) and consult a full report of the evaluation through a **graphical user interface (GUI)**. BOSS [124], which was very similar to ASSYST in its initial specification, later evolved to support Java (a novelty to date) and included a GUI to submit the attempts. Ceilidh [29] gained immense popularity, as it not only granted automated assessment but was also a full course management software, supporting course administration, content delivery, grade weighting, monitoring and recording grades, and typographic (i.e., count of a character or a sequence of characters in the source code) and complexity analysis.

With the rise of the global Internet, the adoption of web-based architectures on new automated assessment systems marks the beginning of the 21st century [47, 75, 104, 123, 141, 158]. Such architecture is characterized by having a client application running on a browser that communicates with an external server containing the functionality and data. The previous existing tools were either adapted, such as Ceilidh (which was even renamed to CourseMarker) [104] and BOSS [123], or were slowly left behind, whereas the proliferation of tools continues to grow [47, 141, 158]. Furthermore, what is assessed and how it is assessed continued to change until 2010 (i.e., before the scope of this review) [112]. Regarding the former, the most popular languages supported were Java, C/C++, Python, and Pascal, and most of the systems were language-agnostic, as their assessment relied on output comparison. Developing traditional programs invoked from the command line, however, is clearly not the common task of a fresh graduate today, and much effort has been devoted to the automated assessment of other works such as GUI development [76, 93, 235], database management [64, 126], assembly programming [147], web development [82, 111, 231], concurrent programming [165], automated testing [9, 72], and diagramming [105, 250]. From the perspective of the teachers, automated assessment tools have also improved. In addition to evaluation reports providing a summary of the assessment, some tools also offer teachers a dashboard with statistical displays about the students' activity (mainly submission results) [75, 141].

The methods of assessing the learners' attempts keep shaping toward long-desired goals, including alleviating the strict assessment of results (e.g., a learner with complete and correct code who only fails to print the result gets 0, if the assessment method used is output comparison), improving feedback, and keeping security during the evaluation (e.g., malicious or wrong code may damage the system). Although the *de facto* approach to assess correctness is compile, run, and compare output (either by printed output or return value), its drawbacks are known and some approaches try to mitigate them by matching through regular expressions rather than direct matching [6]. Nevertheless, the problems remain, as what dictates the grade is only the output, whereas the feedback is also limited to the differences between the expected and the obtained (possibly with some instructor-defined hint rules based on them). Automated testing frameworks typically used in the industry, such as xUnit-based frameworks, are a possible remedy, as (in addition to familiarizing learners with tools they are likely to use in the future) they offer more granularity of assessment and thus are gaining some attention [12, 221]. Notwithstanding, static analysis, which consists of the assessment of a program without its execution and is typically responsible for gathering numerous metrics from source code [5, 197, 236] and detecting plagiarism [3, 191], also seems a promising approach to assess functionality, particularly through some experimental approaches using graph similarity to compare the learner's attempts (dependency graph and abstract syntax tree) to a pool of known solutions [248].

3 RELATED WORK

Several studies have been conducted to review the literature and existing tools for automated assessment of programming assignments. Two studies with particular relevance were published in 2005. First, Ala-Mutka [6] surveyed the features of programming assignments that were automatically assessed up to 2005, including the different techniques and approaches to assess them. Ala-Mutka makes a clear separation between dynamic analysis, which involves the execution of the program, and static analysis, in which the assessment does not execute the program. Moreover, the former is often used to assess functionality, efficiency (e.g., CPU time), and testing skill, whereas the latter is used to check style, syntax errors, software metrics (e.g., X lines of code or $O(n)$ complexity), structural requirements, keyword presence, and plagiarism. Then, Douce et al. [68] reviewed numerous influential systems since the first known automated assessment system in the 1960s to 2005, detailing the earliest systems and introducing recent developments with a focus on tools providing automated assessment of programming assignments. Their work identifies three distinct generations of automated assessment systems: the first, representing initial attempts to automate assessment; the second, encompassing systems managed through a command-line or local GUI; and the third, consisting of web-based tools. Even though the way authors conducted the reviews is distinct, both conclude that automated assessment tools (1) are just support and not a replacement for teachers, (2) pose many restrictions on what is automatically assessable, and (3) have accurate feedback although it is insufficient for pedagogical purposes.

Ihantola et al. [112] conducted a systematic literature review on the research advances on automated assessment tools for programming exercises from the survey of Ala-Mutka to 2010. Similarly to Ala-Mutka [6], it discusses the features and approaches supported by the tools, both technically and pedagogically, rather than specific tools. The features explored include supported programming languages, interoperability with learning management systems (LMSs), tests (i.e., how tests are defined), resubmissions (i.e., how resubmissions are handled), possibility for manual assessment (i.e., after automated assessment, the teacher can perform a type of manual verification), sandboxing (i.e., how security is guaranteed when running students' code), distribution and availability (i.e., if systems are open-source or otherwise available), and specialty (i.e., systems that assess a special type of programming assignments). The authors conclude that (1) security should not rely on teachers' skills (i.e., default security settings should guarantee enough security); (2) tools should be made available even if only partially developed; (3) the most notable advances relate to testing methods, resubmissions, and security; and (4) emerging research would probably aim the integration of automated assessment with LMSs, security during assessment, and assessment of assignments for exploiting security vulnerabilities in web applications.

Another relevant systematic literature review, performed by Souza et al. [220] in 2016, collects and evaluates evidence on tools that support programming assignments. Unlike the previous reviews, this one focuses mainly on the tools, carrying out an initial survey and selection of 30 assessment tools for programming assignments and categorizing them according to three dimensions: assessment type (e.g., manual, semiautomatic, and automatic), approach (i.e., how to trigger the assessment process), and specialty (e.g., contests, quizzes, and software testing). The main contribution of this work is a "guide" for instructors to identify which of the 30 assessment tools best fits their needs, even though there are a few incongruences such as duplicate tools (T13 and T29) with distinct classifications in diverse categories. Furthermore, there are some significant intermediate remarks such as the importance of maintaining a repository of problems, storing submissions' history, reporting statistics to the instructor, and coping with an LMS as a means to reduce the effort required from instructors.

Keuning et al. [130] present another systematic literature review, but exploring a specific feature—automated feedback, which arguably is the most significant one. It surveys tools to support programming learning that give automated feedback, not only automated assessment tools, aiming to discover what kind of feedback they provided, which techniques they use to generate feedback, how adaptable their feedback is, and how these tools are evaluated. The study analyzed and categorized the feedback generation in 101 tools, concluding that the feedback generated is narrow and mainly consists of identifying mistakes. Moreover, the feedback either misses "knowledge on how to proceed" or does not recognize alternative strategies. However, we should also notice that "the upcoming trend of data-driven tutors shows promising results."

This review follows a similar approach to that of Ala-Mutka [6] and Ihantola et al. [112] for the recent (2010–2020) developments of automated assessment. It also distinguishes itself from the mentioned reviews by the features it covers, including exercise types, methods for assessment, security in code execution, feedback, and learning analytics, and offering new perspectives for further research.

4 SCOPE AND METHODOLOGY

From Section 3, we can conclude that for the past decade, the trending features and approaches for automated assessment in CS have not been reviewed widely, except for the existing tools (Souza et al. [220] in 2016) and automated feedback (Keuning et al. [130] in 2019). Hence, there is a research gap concerning the latest proposed techniques, the more effective proposals to implement, and remaining challenges requiring research effort. This review intends to fulfill this gap, answering the next questions about the literature on automated assessment from 2010 to the first half of 2021:

RQ1: Which CS domains beyond traditional programming have exercises assessed automatically?
RQ2: Which testing techniques have been proposed?
RQ3: How secure is code execution?
RQ4: Which techniques to generate feedback from automated assessment to learners have been proposed?
RQ5: Which learning reporting and analytics have been included in these tools?
RQ6: Are the presented techniques and tools effective?
RQ7: What are the expectations for the new decade?

To this end, we have carried out a state-of-the-art review. This type of review, not necessarily exhaustive when compared to a systematic literature review, aims to provide a critical analysis of the extensive literature produced recently (e.g., in the past decade), possibly giving new perspectives on an issue or pointing out topics requiring further research [67]. Hence, we have decided to collect data from conference proceedings and journals indexed by three of the largest bibliographic databases and meta-search engines of scientific publications in the area of CS, namely ACM Digital Library, IEEE Xplore, and ScienceDirect, using a keyword-based search on the full-text. Listing 1 presents the constructed search query. Initially, the search query included the term *computer science* in the last disjunction. However, many entries related to automated assessment of non-CS assignments were included in top results, as the automated assessment itself is a CS field. We have noticed that not including this term results in more accurate entries (i.e., according to our knowledge of publications on the field), including publications on automated assessment of other CS assignments than programming. Hence, the term was excluded. Additional filters to

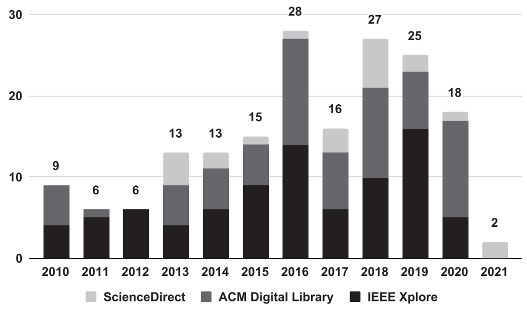

Fig. 1. Identified set of relevant publications by year and indexing database.

limit publication year to values ≥ 2010, publication type to research/review articles in journals and proceedings, and subject area to CS (only applicable in ScienceDirect) have been added.

```
("automatic" OR "automated") AND ("assessment" OR "evaluation" OR
    "grading" OR "testing") AND ("programming" OR "program" OR "
   code")
```

Listing 1. Search string used to query electronic databases

The total results obtained from the conducted keyword-based search include 46,344, 6,738, and 58,786 publications from ACM Digital Library, IEEE Xplore, and ScienceDirect, respectively. Of these, the top 200 publications by relevance (i.e., a measure that considers the amount of citations, cited works, among other facets of a publication) were extracted from each database. From the 600 results collected, 15 duplicates were removed, and the remaining were subject to a pre-processing phase, aiming to identify the relevant publications for analysis. For this phase, we have read the titles and abstracts of the papers and, in a few cases of doubt, performed a fast scanning through their contents to apply the following inclusion/exclusion criteria:

IN if it presents either an automated assessment system or tool for CS education.
IN if it presents either an automated assessment technique or method for CS education.
IN if it presents an experience on the use of automated assessment for CS education.
IN if it presents a review on the use of automated assessment for CS education.
OUT if the automated assessment approach described is only applicable in the industry (based on authors' opinion, if in doubt).
OUT if it is a tutoring system or other system that does not automatically assess CS tasks other than quiz-based tasks.
OUT if it describes a general-purpose automated assessment approach, such as typical quizzes.
OUT if not written in English.
OUT if only abstract is available.

The outcome of this preselection phase encompasses 178 relevant publications (out of 585 publications), which were selected for further analysis. Figure 1 shows the distribution of the relevant publications by year and indexing database.

Unfortunately, this set of publications lacks notable works. In fact, the ineffectiveness of keyword-based literature search in capturing relevant publications has already been demonstrated [113]. To solve this problem, we have conducted a thorough two-way snowballing

Table 1. Results of the Snowballing Process

Iteration	Samples
BASE	178
1st	732 (+554)
2nd	889 (+157)
3rd	899 (+10)

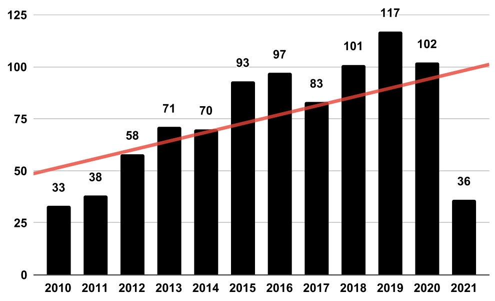

Fig. 2. Final set of publications by year (with the trend line).

process (i.e., including both reference and citation tracking), starting from the previously identified set of 178 relevant publications. To this end, we have adapted a Python script for using the Semantic Scholar RESTful API to fetch citations and references and present their title and abstract, one by one, for us to apply our previously described inclusion/exclusion criteria. Table 1 shows the number of papers found by each iteration of the snowballing process.

The resulting set consists of 899 publications from more than 2,100 distinct authors from all over the world. The composition of this set emphasizes the increasing interest in automated assessment over the years, as depicted in Figure 2, which shows the number of publications per year as well as the trend line.

The selected publications were then subject to a three-round analysis. The first round consisted of the identification and cataloging of the main contribution of each publication. To this end, we re-examined their titles, abstracts, references, and, in a few cases, the contents, assigning tags according to the subjects. Tags range from *Tool*, *Experiment/Validation*, *Feedback*, *Special*, *Learning Analytics*, *Assessment Technique*, and *Security*, which identify the research question, to tags characterizing the novelty presented such as *Open-Source*, *Static Analysis*, *Unit Testing*, and *Containerization*, among many others. This round excluded 121 publications due to poor quality (e.g., incomprehensible sentences or lacking description of the methods/novelty), having no citations and three or fewer references within the collected set simultaneously, or only because the previous rounds failed to apply the criteria. The second round involved a thorough review of the remaining publications (778), summarizing their main contributions and organizing the papers in order of time, according to their developments. Finally, the third round was a verification round to validate the results of the previous rounds. The verification consists of two phases: (1) checking the removed publications for possible exclusion errors and (2) random-sampling the tagged set of publications to validate assigned tags.

5 CURRENT STATE OF AUTOMATED ASSESSMENT IN CS

During the past decade, automated assessment in CS continued as a topic of great interest among researchers. Several approaches have been introduced to assess new types of exercises, new testing techniques were proposed, new possibilities opened to facilitate the execution of untrusted code safely, feedback improved, and, simultaneously, learning analytics is consolidating its importance in CS education. This section introduces the most notable developments reported in the literature review.

5.1 Special CS Domains (RQ1)

Traditional programming exercises that run through the command line are fundamental for novice programmers to acquire practical programming competencies. However, they cannot accommodate every practical skill that a CS graduate must learn. Today, the final product of a development project is often a program tailored to be deployed on the cloud or called through a web or mobile user interface, or a desktop GUI. These products are the result of a long pipeline involving architectural decisions, development, testing (e.g., effectiveness, efficiency, and security), and deployment. A CS graduate must be able to participate in any of these tasks in environments where a small mistake in the solution can cause losses of millions [224]. Hence, the interest in evaluating these skills automatically is often the trigger for a completely new automated assessment approach or tool.

There are several works published in the recent literature that specialize in assessing other CS skills than traditional programming. The ones identified in this review are presented in the following, according to the skills of CS that they cover.

Visual programming. Visual programming has proven to be an effective means to introduce programming at different educational levels [127]. Hence, the interest in assessing it automatically is evident. The proposals have emerged in two categories: static analysis and dynamic analysis. Static analysis approaches either check command usage or compare the student's program with a model solution [37, 146, 157, 169, 232, 246] aiming to assess mostly the computational thinking, whereas dynamic analysis approaches are applied when the problem has a single solution and thus they compare the final program state to the expected output [63, 122].

Programming assignments (alternative). Programming exercises are not limited to blank sheet exercises, where the student is challenged to solve, from scratch, a presented problem statement. An alternative type of activity can foster new competencies, such as understanding the code developed by others and debugging, or be more suitable at different phases of students' learning path. For instance, Kramer et al. [134, 135] proposed the task of highlighting elements in a given source code and investigate several measurements that could be used to evaluate a response to this task automatically. Hakulinen and Malmi [96] presented a method that applies QR (quick response) codes (i.e., tasks are formulated so that the correct output is a QR code, which can be tested by scanning with the mobile). Jin [121] reports on a web-based system based on `pythontutor.com`, an open-source code visualization system that graphically illustrates the execution flow and state changes of a full program, which allows instructors to set up full tracing exercises with automatic grading.

System administration. Tasks of a system administrator comprise the maintenance, configuration, and reliable operation of computer systems. Whereas traditional automated assessment of programming assignments involves a single file (or folder), assessing a system administration task may require checking the complete operating system. However, some solutions have been presented, such as an online grading system to grade and provide feedback on assignments using live **virtual machines (VMs)** [24] and a framework for automatic evaluation of Linux-based operating system exercises, named *LINSIM* [42].

Formal languages and automata. Formal languages and automata are core components of the undergraduate CS curriculum, and many simulation tools [46] have been developed to make their learning more engaging. One of such tools is JFLAP [92], which has been enhanced recently to automatically test and give feedback to students' answers to some generated problems [209]. In a similar line of research, Alur et al. [11] presented a technique for automatically grading and providing meaningful feedback to deterministic finite automata (DFA) constructions. More on formal languages, two web-based tools to automatically mark and give feedback on conversions from natural language to first-order logic [175] and first-order logic to clause form [94] have been proposed.

Software modeling. Models are forms of description with which we can abstract details to represent and communicate what is important about a complex system. Visual models are commonly used in software development not only to plan and visualize different aspects of the software before (during and after) it is implemented, but also as programming languages, which are becoming increasingly popular to introduce novice programmers [162]. The automated assessment of software modeling and visual programming is, thus, a research focus empowered by the existence of multiple correct solutions. On the software modeling side, the proposed solutions are mainly for UML (unified modeling language), such as class diagrams [34], activity diagrams [228], use-case diagrams [239], general purpose [57, 200], and extensions to UML class diagrams [38], but there are also solutions for Entity-Relationship (ER) diagrams [57, 211]. The evaluation is typically done by graph similarity between a model solution or a set of model solutions.

Software testing. Students might develop wrong programming behaviors, such as (1) adopting trial-and-error approaches without thinking between attempts, (2) assuming that a program that compiles without complaining has no errors, and (3) rashly considering a program producing the expected output on a few tests as correct [19]. To mitigate these issues, a few studies have suggested the introduction of testing into CS curriculum [22, 73]. Testing activities challenge the students to demonstrate the correctness of a program by developing tests and then assess how well they achieve this goal. Hence, teachers have to evaluate not only the developed program but also the test suite. Fortunately, there are already automated assessment tools to handle this task. For instance, ProgTest [65] incorporates coverage testing tools as a support for applying test criteria and evaluating the coverage of the test suite. Prof. CI [155] uses continuous integration services to automatically assess the progress of participants and incrementally assign the next small task to foster test-driven development.

Database. Querying and manipulating databases is a daily task for many developers, and the source of major performance bottlenecks and security concerns [49, 222]. Therefore, it is important to offer learners sufficient practice in using them. Common automated assessment systems relying on output comparison can use custom scripts to send the query to an SQL server, receive its results, and compare to the expected output. However, that involves much configuration and lacks adequate security and feedback. Kleiner et al. [133] presented a plugin—aSQLg—for automated assessment and feedback of student-submitted SQL queries that can be used in conjunction with Web-CAT [75], as well as any proprietary course management system. Panni and Hoque [171] introduced a model that compares an SQL answer with the reference solution and calculates a score based on the syntactic similarity between the two.

Web development. The interest in the automatic assessment of web-based assignments is not new, as it is part of the curriculum of CS (at least as an optional subject), as well as hundreds of courses available online almost since the Web 2.0 era. However, assessing a web project involves not only the validation of the source code but also functional user interface testing in the browser, simulating actions on the content, and checking the respective changes to the browser state. Moreover, assignments may use several different services from web servers to databases, running concurrently and possibly exposing ports for external access. Hence, diverse solutions continue to be proposed. The WebWolf framework [214] allows instructors to write JUnit-like methods simulating user actions and to do assertions on the browser state to test websites, leveraging on the Selenium automation framework to drive a headless web browser. Similarly, Peveler et al. [179] presented a system that only differs from WebWolf in that it utilizes Docker and its concept of containers to run the student's code and required services (providing isolation) and allows the instructor to grade manually.

Parallelism and concurrency. Parallel and concurrent programming are hard to assess manually, as two or more processes run simultaneously (or in an interleaved fashion through context switching), and thus multiple execution flows need to be considered when looking at the source code and the used resources. Hence, it is even more demanding to support automated assessment on these tasks. Aziz et al. [20] presented a plugin for Web-CAT [75] to assess and generate feedback on parallel programming assignments. Another proposed alternative is SAUCE [110], a web-based tool for automated assessment of programming assignments with specific features designed for the teaching and evaluation of parallel programming using C++11 threads, OpenMP, MPI, and CUDA programming models.

Mobile development. The availability of mobile devices (e.g., smartphones, tablets, and smartwatches) and their prevalence in routine Internet-based activities increases the market demand for mobile development skills and thus the need to teach them in CS education [43]. Unfortunately, mobile applications are also challenging to develop due to their commonly complex GUI and advanced functionality, including reading data from sensors and communicating with remote services. These are difficulties passed on to their evaluation, which is usually done manually in educational contexts [230]. To the best of the authors' knowledge, RoboLIFT [8]—a library to facilitate unit testing of Android applications—was the first attempt to automatically assess mobile-based applications submitted by students. Recently, a few tools resulting from joint efforts with companies have been proposed to assess mobile applications automatically [41, 153]. First, Madeja and Poruban [153, 154] reported on a testing environment for Android applications based on three types of tests: unit, integration, and user interface tests. Later, Bruzual et al. [41] presented a system for automated assessment of Android exercises with cloud-native technologies, which leverages on a mobile app testing framework largely used in the industry instead of custom testing libraries. In a separate branch, the popularization of MIT App Inventor in CS education motivated some work on the assessment of the interface design of Android applications developed with it [217, 218].

Computer graphics. Computer graphics assignments pose a novel challenge to automated assessment. Instead of receiving a text-based output that can be compared to the expected output, the output is an image or animation often having numerous correct variants. Notwithstanding, there are a few works addressing this challenge. Sanna et al. [208] proposed a method to assess 3D models by using three metrics to measure the closeness of an attempt to a given reference solution: (1) the mesh complexity (i.e., number of polygons); (2) mesh similarity (e.g., using a method designed for image retrieval); and (3) material similarity by computing the differences in the hue, saturation, and value (HSV). This work was later extended to animated 3D models [138]. Regarding OpenGL, Hodgkinson et al. [106] presented a new tool for the automatic assessment of assignments—glGetFeedback—that uses invariants and intermediate data obtained from the OpenGL state machine and the rendering pipeline, whereas Wünsche et al. [251] introduced a new type of question for CodeRunner [151] that handles the automatic assessment of OpenGL programs.

5.2 Testing Techniques (RQ2)

The testing technique is the core of an automated assessment approach, as it defines the boundaries of what is assessed, how feedback is produced, and which are the necessary security measures. The single requirement of a testing technique is that a type of measurement value can be extracted from the submitted program and compared to the assignment requirements or given solutions [6]. In this review, the gathered testing techniques and tools have been categorized by the program feature they evaluate. The results are presented in the following.

Functionality (outcome). Testing whether a program functions according to the given requirements is the most common approach to grade programs. Traditionally, these tests consist of running the program on a set of test cases, having the input to feed the program and the expected output to which the obtained is compared to [6, 61, 77, 241]. This is called the **output comparison (OC)** method. However, the comparison strictness is not adequate to learning settings because a single white-space character causes a completely wrong submission. This has motivated several attempts to alleviate it, such as partial grading with pattern matching to assign a grade even in the presence of printing formatting issues (e.g., trailing and excess spaces, letter-case matching issues, space/tab misuse, and incorrect sequence order) [148], regular expression matching [183], and comparing output tokens instead of characters [255].

Industry-based testing tools, which are actively developed, robust, and cover almost any programming language and application domain, are tailored for checking the functionality of programs, and thus their adoption in educational domains comes without surprise. Among the most widespread, there are the language-specific xUnit family of testing frameworks (e.g., JUnit for Java, CUnit for C, PyUnit for Python, and HUnit for Haskell), which derive their design from a unitary testing framework for Smalltalk—SUnit [25]. Comparing to OC, **unit testing (UT)** offers more granularity of evaluation, being able to assess classes, methods, and even statements [6]. Examples of automated assessment tools using xUnit testing frameworks include JPLAS [83] and STAGE [172]. Web testing frameworks, such as Selenium [214] and Mocha [91], and mobile testing frameworks, such as Appium [80], are also applied to check the functionality of students' web pages and mobile apps automatically.

Functionality (source code). Evaluating the functionality of a program only looking at the outcome (i.e., through *blackbox* techniques) is not the best approach for pedagogical means. Recently, *whitebox* techniques for automated assessment of functionality have been emerging. For instance, JavAssess [115] is a Java testing library that makes use of reflection in combination with other meta-programming features to analyze, mark, and manipulate the student's source code at runtime. However, this is a rare case of dynamic analysis using *whitebox*.

Static analysis has been gaining popularity in the assessment of program functionality. The common approach is to parse the student program and construct an abstract syntax tree, transform it into a graph representation, and calculate the graph similarity measure on a pool of model graph representations. The closest model graph representation (or cluster) is used to compute the differences, if any. For instance, Naudé et al. [160] followed this approach using a system dependence graph for graph representation, whereas Zougari et al. [258] and the prototype tool eGrader [10] applied it only after a xUnit-based testing step to guarantee that incorrect submissions are not marked as correct, using a control flow graph and a combination of control and system dependence graphs as the graph representations, respectively. LAV [247] combines this approach based on the control flow graph representation with bug finding and traditional *blackbox* testing techniques to estimate a composed grade.

The challenge of comparing through the structural similarity of programs is the multitude of solutions available for the same programming assignment. Behavioral similarity obtained through symbolic execution can help mitigate this issue [18, 145]. Symbolic execution consists of analyzing a program to determine what inputs trigger each part of the program, assuming symbols for the inputs rather than obtaining actual inputs as normal execution would [21]. Arifi et al. [18] proposed a method to calculate the grade of the behavioral similarity value toward a reference program. Li et al. [145] explored three different metrics—random sampling, single-program symbolic execution, and paired-program symbolic execution—to approximate the behavioral similarity between programs and describe its application in automatic grading, students' progress indication, and hint generation. AutoGrader [149] searches for semantically different execution paths, captured using the weakest preconditions and symbolic execution, between a student's submission and a reference implementation returning a counterexample, if it exists.

Furthermore, symbolic execution has other relevant applications for functionality checking, particularly to furnish a test suite that adequately covers most edge cases of a programming assignment. For instance, Song et al. [219] combine symbolic execution and enumerative search to detect logical errors efficiently in functional programs, generating a counterexample if a bug is found. The authors compare the technique against property-based testing, a well-known approach for testing functional programs, achieving improvements in detection and efficiency. Another example is GRASA (GRAding-based test Suite Augmentation) [168], which detects and clusters incorrect submissions through paired-program symbolic execution behavioral equivalence, then generates a minimal set of additional tests exposing the issue to augment the existing test suite.

Code quality. Most of the development costs of a software product are incurred during the maintenance phase, which is where code quality gets more evidence [132]. Hence, it is important to educate students with knowledge about code quality. There are several tools used in software development that are frequently integrated into automated assessment tools to measure code quality, such as Checkstyle [101, 117, 150], PMD [101, 185], FindBugs [101, 150, 166] or SpotBugs [101, 223], and SonarQube [150, 207]. These tools are designed to find common programming flaws including unused variables, empty catch blocks, and unnecessary object creation, in the main programming languages.

Nevertheless, there are specific issues that affect mainly novice programmers and better ways to report them in learning contexts. Hence, some tools have been developed. WebTA [238] can identify and flag certain types of antipatterns, allowing for customizable critique triggers and messages. A different perspective is followed by Gerdes et al. [89], who used a language to describe strategies [102] in combination with programming transformations to recognize the strategy utilized in different student programs by comparing with the strategies of a limited set of model solutions. As these strategies encode the expected good programming practices, the student program should match with at least one of them, as otherwise she is likely to be following bad practices.

Software metrics. Software metrics encompass other general but relevant measurements of a program, such as requiring a program to have an algorithmic complexity lower than or equal to $\mathcal{O}(n)$. Tools generally permit the creation of custom scripts for these purposes without explicitly providing support to assess them [180, 203, 227]. Notwithstanding, JavAssess [115] can check (using reflection) the cyclomatic complexity and the number of usages of a certain construct, among others. Benac Earle et al. [27] used algorithmic complexity (computed using property-based testing), the number of noncommenting source statements, and McCabe's cyclomatic complexity as weights of the grade assigned to a program.

Test development. Modern software development practices promote test-first approaches, as they help to reveal design flaws early in the development process and ensure (by warning) core functionality persists between releases [71]. Traditional automated assessment approaches discourage the development of tests, as the tools already verify the correctness of the program. However, students should learn how to design and test their programs [72, 225]. Hence, a few automated assessment tools evaluate the quality of the test suites produced by students, particularly through test coverage (i.e., the amount of code executed when it runs). External tools are typically integrated to measure coverage such as CodeCover [172, 174] and Emma [180, 205].

Since test coverage is not a reliable metric (i.e., executing a statement does not mean checking if it is correct), Aaltonen et al. [1] proposed mutation analysis as a replacement of test coverage. Mutation analysis consists of seeding simple programming errors into a correct program, creating mutants, and running the test suite on each mutant. The test suite that recognizes more defects is better. In the same line of using the concept of mutation analysis, Clegg et al. [54] developed a tool to help instructors prepare more accurate and fair test suites to cover students' misconceptions and grade them appropriately (although intended for functionality testing, the technique harnesses test suite quality through coverage of mistakes). Dewey et al. [66] presented a technique for evaluating a test suite by weighing its ability to discover defects with that of a test suite automatically generated using constraint logic programming.

Plagiarism. Automated assessment and plagiarism have been connected since the early days for obvious reasons [6]. Therefore, most of the recent tools already provide it out-of-the-box [50, 164, 255] or are able to integrate one of the existing specialized tools [163]. The methods used by these tools for detecting plagiarism in source code can be broadly classified into three categories, according to what aspect of source code they compare: structure, semantic, and behavior.

Structural approaches compute similarity by matching identical structures in source code between plagiarism-suspected pairs. These structures can range from token sequences extracted from source code [4, 16, 114] to advanced data structures representing the structure of source code (e.g., trees) [81, 257]. The most common kind of this approach consists of measuring the similarity between lexical token sequences through techniques such as cosine similarity [114, 167], token edit distance [16], Running-Karp-Rabin Greedy-String-Tiling [26, 125, 186, 229, 259], Winnowing [69, 259] (introduced in MOSS [4]), and local alignment [196]. Other approaches use the programming language grammar to devise parse trees or abstract syntax trees for comparison [81, 244, 257].

Semantic approaches measure how similar the meaning of plagiarism-suspected source codes are. These methods work similarly to structure-based ones in that they start by extracting a representation of the source codes and then compare them. However, they use semantic information that is not denoted through the grammar of programming languages. Thus, they typically encode information using graphs, such as program dependence graphs [52, 210], control flow graphs [45], and call graphs [187], rather than tokens and trees. Other semantic approaches involve analyzing the similarity of relationships between terms through latent semantic analysis [23, 58, 237].

Behavioral approaches are based on the assumption that the runtime behavior of a program can uniquely identify it, and thus similar programs have identical behaviors. Therefore, this kind of approach requires dynamic analysis of the programs. Several distinct techniques exist for comparing programs' behavior, differing in which dynamic aspect they observe. For instance, there are techniques analyzing data at runtime [48, 120], interactions with the execution environment [14, 48], functionality [145], and logic [152, 256].

5.3 Code Execution (RQ3)

Dynamic analysis is the most common and effective form of assessing the functionality of a program. However, the execution of code written by learners (or any other external user) demands strong security measures, as it may provoke service outages or even damage the underlying system.

Table 2. Characteristics of the Different Security Models (Table Extended from Sims [212])

	Resource Limitation					Security		Misc		
	CPU	**File System**	**Memory**	**Processes**	**Network**	**Access Control**	**SU Required**	**Idempotency**	**Dependencies**	**Overhead**
User Permissions	YES	YES	YES	YES	YES	YES	YES	YES	NO	LOW
Process Jails	NO	NO	NO	NO	?	YES	YES	NO	NO	LOW
MACs	?	?	?	?	?	YES	YES	NO	NO	LOW
JVM	NO	YES	YES	YES	YES	YES	NO	NO	NO	LOW
VM	YES	YES	YES	YES	YES	YES	NO	YES	YES	HIGH
Containers	YES	YES	YES	YES	YES	YES	NO	YES	YES	MEDIUM

A question mark (?) indicates that the limitation depends on the implementation.

The numerous possible ways in which either a malicious or an erroneous program may harm the system [78] make it a risk even for closed institutional tools. Therefore, some solutions have opted for either running the code in the client, which has obvious assessment flaws (e.g., tampering with the result before sending to the server), or excluding keywords/statements from the source code using a type of blacklist, which is also flawed (i.e., there is no such limited list) [83, 112]. A secure test environment must at least do the following:

- Prevent access to sensible test data, such as output data;
- Block any attempt to leak test data to the outside;
- Guarantee the idempotency of different test runs—that is, two or more runs of the same test must execute under the same conditions, independently of the order;
- Limit access to the system resources, mainly CPU time, the amount of system memory, filesystem, number of child processes, and network connections.

There are three generic security models for addressing these requirements: user-level restrictions (i.e., using Unix user and group access controls and resource limitation mechanisms), process-level restrictions using process jails and **mandatory access controls (MACs)**, and virtualization (i.e., running a virtual system in a layer abstracted from the actual hardware). The latter has three common types: the **Java virtual machine (JVM)**, a virtual stack machine provided by the Java runtime that executes Java bytecode, confining Java processes to a limited virtual environment; the "standard" VM, which refers to an entire operating system hosted as a guest application; and containers, which are a lightweight form of handling virtualization that leverages features of the operating system to isolate processes and control their access to system resources. All models offer unique ways to isolate the compilation and execution of unknown code, and each has its (dis-)advantages over the others. Table 2 presents a summary of the characteristics of the different security models, including the resource limitations they can constrain, whether they provide access control, whether they need superuser privileges, whether the tests are idempotent, whether dependencies (compilers' versions, packages, etc.) are also isolated, and the amount of overhead they imply.

Furthermore, the weaknesses of each model must also be considered. Even though user permissions seem to fulfill the mentioned requirements while having a simple configuration, this model trusts the whole underlying codebase, including dependencies, requirements of programming languages, and system packages. Any vulnerability in this chain that permits the program to escalate privileges would completely break the security of the system. A similar situation happens with process jails. Regarding MACs, the main disadvantage is that they have a highly complex configuration whose failures may have severe repercussions. On the virtualization side, the main drawbacks per type are JVM, which targets Java-specific projects; VM, which has a high startup cost; and containers, which are less secure than VM and more costly than others (except VM).

Although most tools still rely on user permissions [2, 60], jails [40, 164, 203], and JVM [115, 172, 242] security mechanisms, several new tools [156, 180, 226] are adopting containerization, particularly through Docker containers, to perform the dynamic analysis of students' programs.

Docker reduces the entry barrier to using containers, which enables their fast-growing popularization lately, as containers also offer a similar level of isolation to that of a VM while keeping a much lower overhead, slightly above other solutions depending on the image [178]. In addition to that, being able to run the code in a clean environment with only the specific version of the language compiler and libraries needed allows great flexibility and reduces compatibility issues. Although there is no declared use of MACs as the primary security layer for untrusted code execution, which is reasonable due to their high setup cost, VMs have only a few known uses [194, 201] due to their initialization cost. Notwithstanding, VMs and other real machines are usually applied as a secondary security measure for running the students' programs separately from the rest of the system for additional security [40, 70].

5.4 Feedback (RQ4)

Feedback provides learners with information on the quality of the produced work, compared either to the expectations or adequate standards of the mentioned work [39]. While summative feedback, either in the form of nominal classifications, numeric grades, or binary results, is adequate and useful for assessment, formative feedback aims to correct learners' understanding of the problem to improve their solution and thus demands more information. Hence, feedback yields a strong impact on learning progress and achievement, but its influence can either be positive or negative depending on its type and the way it is delivered [99].

According to Keuning et al. [130], there are five types of feedback in learning tools for programming that focus on generating improved work. They are described as follows:

- *Knowledge about task constraints* (KTC), which includes information about the task, such as requirements or general processing rules.
- *Knowledge about concepts* (KC), which encompasses explanations as well as illustrations about the concepts being taught, generated while a student is working on an exercise.
- *Knowledge about mistakes* (KM), which contains details about the mistakes, including failed tests (TF), compiler errors (CE), solution errors (SE), style issues (SI), and performance issues (PI).
- *Knowledge about how to proceed* (KH), which provides learners with instructions on how to address their mistakes and take the next step toward a solution.
- *Knowledge about meta-cognition* (KMC), which consists of feedback to check if a student knows why an answer is correct.

Although giving KTC and KC feedback can be important in automated assessment tools for CS assignments, these feedback types are too dependent on the tasks' definition, being more a matter of configuration, either to the instructor or exercise author, than a research need. Furthermore, as most of the assignments considered are open-answer, these tools are not commonplace to incorporate KMC feedback, even though a few recent attempts exist [142, 188, 189]. Hence, research has concentrated efforts on KM and KH types of feedback. Nevertheless, the results summarized here are related to the advances on KH, as Keuning et al. [130] already covered automated assessment tools for programming assignments, concluding that they mostly offer test-based feedback while very few give KH feedback.

Feedback on how to proceed should consider the current state of the learner's attempt to generate personalized guidance to help them overcome the next barrier toward a solution. It involves either of the following:

(1) Recommending a possible correction when the learner encounters a bug in the code,
(2) Giving a hint on the next step to take when the learner is lost, or
(3) Suggesting an improvement if the solution could be better, regardless of the correctness.

Regarding the guidance of type 1, novel approaches commonly suggest the application of techniques such as semantic representation, program synthesis, and symbolic execution in combination with error models, fault-localization techniques, and **automated program repair (APR)**. As an example, Singh et al. [213] introduce a tool that reasons about the semantic equivalence of student programs with reference implementations to derive an error model describing the potential corrections and then combines that model with Sketch-based constraint-based synthesis [215] to compute minimal corrections to incorrect programs. This work assumes the existence of (at least) one correct solution to apply techniques relying on code similarity measurements and the predictability of students' errors to narrow the search-space for a fix. Other approaches build on these assumptions [55, 131], with distinct methods (e.g., plagiarism-based clustering [55] and symbolic- and dynamic-based clustering [131]), to cluster incorrect programs by their "error class" and send feedback to the resulting clusters instead of all incorrect programs separately. Even though these last techniques do not recommend a correction directly, they have been considered KH feedback because they pinpoint the potential fault locations and provide suggestions that are sufficient, in the authors' opinion, for a novice programmer to address them.

APR, an emerging technology that fixes software bugs automatically, is a growing trend among researchers. APR, formerly designed to correct large-scale industry software, only requires a test suite to conduct the repair process, which consists of changing the original program to make failing tests pass. This technique, however, requires several adaptations for formative settings as programs from a student are often severely incorrect leading to lower repair rates and providing a fully correct solution is not proper formative feedback [254]. For instance, Bhatia et al. [33] proposed a two-step approach that, given a student program, first, uses a set of submissions without syntax errors to construct a recurrent neural network (RNN) model of token sequences to hypothesize possible fixes to existing syntax errors. After all syntax errors are resolved, the second step uses constraint-based program synthesis [216] to find the set of minimal changes to the program that make it functionally equivalent to a reference implementation (search-based correction techniques are only applicable because the programs are tiny, i.e., lines of code ≤ 20). Another example is LoopFix [249], an APR for loop bugs that leverages existing bug localization techniques to rank buggy statements and then exploits a component-based program synthesis [119] approach to synthesize a patch based on the runtime information obtained through symbolic execution.

The previous APR-based approaches limit the search space of edits to be navigated, the former [33] by considering only tiny programs with less than 20 lines of code and the latter by focusing on loops. Hu et al. [109] proposed an APR method that limits the search-space by matching with a similar correct solution. First, each solution of a pool of available correct solutions (at least one) is refactored to a semantically equivalent solution. Then, given an incorrect program, it matches the program with the closest matching refactored solution based on their control flow structure, infers the input-output specifications of the incorrect program's basic blocks from the execution of the correct program's aligned basic blocks, and uses them to modify the blocks of the incorrect program via search-based synthesis. Similarly, Clara [95] clusters the existing correct student solutions by control flow similarity, matches a given incorrect attempt to one of the clusters, and generates a minimal repair of the differences. Nevertheless, fixes generated by APR do not address the underlying misconceptions (i.e., provide a bug correction but not much more information) and may not be stylistically correct [100]. Head et al. [100] presents an approach to send teacher-produced feedback to verified corrections rather than just the bug correction. First, it applies a data-driven program synthesis technique that learns code transformations (i.e., sequences of rewrite rules applied to the abstract syntax tree of a program) from examples of bug fixes. Then, it clusters incorrect submissions by necessary transformations and asks the teacher (semi-automatic) to validate them and give feedback. Finally, it propagates the resulting feedback to all incorrect submissions that are fixed by the same transformation.

Guidance of type 2 is more common in programming tutors, yet similar approaches may fit automated assessment tools. For instance, AutoTeach [15], an incremental hint system for automated programming assignments, displays a series of incrementally revealing suggestions and details of the expected solution upon request from the student, while providing automated assessment based on a set of unit tests. Hints are written by teachers within the solution code through annotations with special processing directives. Numerous techniques for generating hints to support the learning of programming have been developed, ranging from those leveraging on feedback previously added by instructors [97, 100] to those extracting patterns from peer data [139, 140], generating personalized paths to solutions [129, 202], and combining a few of these [181, 192].

With regard to guidance of type 3, Araujo et al. [17] present a technique that measures code quality by comparing the learner solution to a baseline solution provided by the teacher, according to several metrics including logical lines of code, the Halstead volume [98], cyclomatic complexity, and adherence to coding standards. These measurements are used to provide suggestions on what could be improved (e.g., "your program has too many loops").

5.5 Learning Analytics (RQ5)

The measurement, collection, analysis, and reporting of data about learners and their context for understanding and optimizing learning, formally known as learning analytics, is a topic of great interest among researchers, especially in CS education, due to inherent challenges in learning and teaching programming [90]. Understanding students' thinking, difficulties, study habits, and trajectories throughout a learning activity can help instructors to identify better scaffolding strategies, refine assessments, offer rich and tailored feedback to students in difficulty, and have some insight into good and bad behavior patterns [36]. Early identification of learning issues has significant impact on the learning progress, as it enables timely assistance [59] that would otherwise only happen after assessment. Continuous assessment, which is facilitated thanks to automated assessment tools, can help to mitigate this problem. Undeniably, there is a great amount of valuable information not only in the final response of a student to an open-ended question, such as a traditional programming assignment, but also in the whole development process of that response [182]. However, means to automate data collection and analysis are required to perform large-scale studies, analyze such minor details of students' behavior, and collect data from interactions outside of class.

Existing tools supporting automated assessment tend to overlook the instructor's perspective, only providing a few mechanisms to collect and visualize learners' data, such as submissions' history, evaluation reports, and completed activities [40, 85]. Other tools [151, 164, 203, 226] integrate with an LMS to offer the instructor a familiar dashboard that, even though it has several analytical and reporting tools for learning [253], has no means to work with the specific data produced in computer programming. There are some tools, however, already investing high in learning analytics. CloudCoder [108] and TestMyCode [194] store keystrokes of the students during the development of their solutions. This granularity level in the collection reduces the amount of lost data, whereas its analysis reveals insights that submissions or even snapshots would not [243]. ProgEdu [50, 252] keeps a history of changes to the source code using Git and stores logging data to provide instructors with various statistics charts from which they can comprehend whether submissions are on time and meet requirements, the distribution of failure types, and the quality of students' code. PRogramming tUTOR (PRUTOR) [62] takes code snapshots at regular intervals and provides visualizations of the code progression of students' programs and how the students approach the programming assignments. Edgar [156] provides a dashboard with multiple visualizations on learners' behavior and assignments.

Furthermore, the offline analysis of data collected by these systems already enabled some promising conclusions, making even more demanding the use of learning analytics in real time. For instance, Chen et al. [51] used log data related to time, effort, and results on homework assignments, captured with ProgEdu, to characterize the learning behavior of students engaged in a computer programming course, leveraging on exploratory data analysis, clustering, classification, and data visualization techniques. This study concluded that students who made more effort to complete their homework early were likely to finish earlier and receive better results, learning motivation was strongly correlated with final grades, and performance was positively correlated with coding style. Moreover, they propose a prediction model for students at risk, which revealed encouraging accuracy. Munson and Zitovsky [159] gathered a number of metrics, such as the number of hours spent coding, number of days worked, ratio of syntax errors, dropout rate, normalized file size, and time spent working at home, from 3-week log data to build prediction models that could explain the major share of variance in final grades in a continuous scale. Leppänen et al. [144] analyzed keystroke data from TestMyCode [194] and found that a number of short pauses of 10 seconds and up to 4 minutes has a negative correlation with exam scores. Chow et al. [53] present an approach for automating the generation of hints by applying a range of techniques such as filtering, clustering, and pattern mining on previous student submission data. This approach has been validated, both quantitatively (i.e., it can generate hints for students 90% of the time from as little as data from 10 past students) and qualitatively (i.e., experts rated positively the usefulness and relevance of the hints), with data collected from the Grok Learning platform (submission-level granularity). A wider overview of the body of knowledge regarding the use of learning analytics for the learning of programming has been performed by the Working Group Report of the 2015 ITiCSE [113].

Recently, a standardized format for logging programming process data—ProgSnap2—arose from the requirements, experiences, and datasets of a large working group of researchers in the field of computing education [193]. This work addresses one of the most significant gaps in the learning analytics of automated assessment data, which is the lack of a standard format to collect, share, and analyze data produced in automated assessment tools.

5.6 Tools

The previous sections focused on the techniques and approaches reported in the literature. Nevertheless, these techniques and approaches are mostly experimental, only implemented in a closed prototype. This section inspects the most recent automated assessment tools of CS assignments published in the literature regarding the features captured previously. The survey captured 128 automated assessments tools. For the sake of simplicity, 98 publications describing a tool have been left out of this review. These include 60 publications that either focused on the same tool, were missing the description of most of the features we aimed to capture, or were considered irrelevant, as no further link to them could be found online (i.e., they were only a prototype). From the remaining 68, we selected 10 for their high and recent development activity (Submitty [180], Edgar [156], CodeOcean [226], ProgEdu [50], CodeRunner [60], Aristotle [2], Jutge.org [176], CodeWorkout [74], TestMyCode [194], and VPL [203]), 5 for being active for a long period of time (Kattis [77], JACK [30, 136], Mooshak [170], URI Online Judge [32], and Web-CAT [75]), and 15 picked in a random sampling step.

A total of 30 tools were included in this review, consisting of 21 web-based platforms (WP), three Moodle plugins (PG), two web services (WS), two cloud-based services (CD), a toolkit (TK), and a Java library (LB). Most of these tools support multiple programming languages, whereas the rest target either C or Java. More than half of the tools are open-source (OS), whereas half of the remaining are available (A) online for demonstration purposes. To better incorporate e-learning systems, a few web-based platforms implement the Learning Tools Interoperability (LTI)

Table 3. General Information about the Selected Tools

Tool/Reference	Type	Language	Availability	Interoperability	URL
Web-CAT [75]	WP	Multi	OS	LTI + Eclipse + BlueJ	https://web-cat.github.io/
PASS [255]	WP	C/C++, Java	A		https://www.cs.cityu.edu.hk/~passweb
Kattis [77]	WP	Multi	A		https://open.kattis.com
ECSpooler [13]	WS	Multi	OS		https://github.com/collective/ECSpooler
STAGE (CodeCover) [172]	PG	Java, COBOL	OS	Moodle plugin	http://codecover.org
JACK [30, 136, 227]	WP	Multi	A	LTI + Eclipse	https://jack-demo.s3.uni-due.de
URI Online Judge [32]	WP	Multi	A		https://www.urionlinejudge.com.br
CodeAssessor [241]	WP	C/C++			
VPL [203]	PG	Multi	OS	Moodle plugin	https://vpl.dis.ulpgc.es/
CloudCoder [108]	WP	C/C++, Java, Python, Ruby	OS		https://github.com/daveho/CloudCoder
CodeRunner [60]	PG	Multi	OS	Moodle plugin	https://github.com/trampgeek/moodle-qtype_coderunner
JPLAS [83–85]	WP[1]	Java		Eclipse	
LX [61]	WS	Multi		Modified Moodle	
Testovid within Protus [242]	WP	Java	OS		https://github.com/ivanpribela/testovid
TestMyCode [194]	WP	Multi	OS	NetBeans	https://github.com/testmycode/tmc-server
CodeWorkout [74]	WP	C++, Java, Python, Ruby	OS	LTI	https://github.com/web-cat/code-workout
Arena [184]	WP	Imp/OO		Git	
JavAssess (ASys) [115]	LB	Java	OS		http://personales.upv.es/josilga/ASys
CodeOcean [226]	WP	Multi	OS	LTI	https://github.com/openHPI/codeocean
Jutge.org [176]	WP	Multi	OS		https://github.com/jutge-org
GradeIT (PrUTOR) [62, 173]	CD	C	A		https://prutor.ai
neoESPA [148, 150]	WP	C/C++, Java, Python	A		http://neoespa.pusan.ac.kr
Enki (Mooshak) [170]	WP	Multi	OS	LTI + Eclipse	https://mooshak2.dcc.fc.up.pt
ProgEdu [50, 252]	CD	Java	OS	Git	https://github.com/ProgEdu-Organization/ProgEdu
Aristotle [2]	TK	C/C++	OS	Github Classroom	https://github.com/mdadams/aristotle
Submitty [180]	WP	Multi	OS		https://github.com/Submitty/Submitty
Codeflex [40]	WP	Java, C#, Python, C++	OS		https://github.com/miguelfbrito/codeflex
Dante [70]	WP	C	A		https://dante.iis.p.lodz.pl
Coderiu [28]	WP	Multi	OS		
Edgar [156]	WP	Multi	OS		https://gitlab.com/edgar-group

[1]Supports offline mode.
WP, web-based platform; PG, plugin; WS, web services; CD, cloud-based services; TK, toolkit; LB, library.
OS, open-source; A, available online.

specification to sign in students using the LMS and maintain an updated gradebook [195]. There are also tools providing a plugin to integrate either with Eclipse or NetBeans IDE (Integrated Development Environment) and relying on Git workflows. Table 3 presents general information about the selected tools, including either the URL to their source code or homepage (if available). Tools are sorted by their release year, according to all information we could gather online about them.

Even though inadequate for learning contexts as they only verify the outcome, OC and UT are the techniques upon which these tools typically rely, as functionality is widely the most looked up measure and their accuracy for that is indisputable. New forms of assessing functionality through the source code are still not mature enough, and thus they are applied only after checking that the solution is not correct, to generate hints on how to correct the program (e.g., [173]) or assign a partial grade (e.g., through Java Reflection (JR) [115]). Plagiarism, code quality, and other software metrics are typically either handled by external tools, such as JPLAG [190] and CheckStyle [117], or left for the tool manager to set up with custom scripts. The format of a custom script depends on the tool, but it is generally a standard script with the single constraint that it returns a grade and/or feedback. Table 4 presents the testing methods used in the selected tools.

Evaluating the functionality based on the outcome requires the execution of students' programs. Therefore, the security model for untrusted code execution is one of the design decisions that must be taken into account when considering such tool. The user permissions model is the most usual approach, as it delivers sufficient security, particularly when the assessment happens in an external machine, with a lower setup cost, whereas JVM is the model chosen for tools providing support only for Java. Notwithstanding, containers already make almost 25% of the security models. Table 5 outlines the security mechanisms applied in the selected tools for untrusted code execution.

Table 4. Summary of Testing Methods Used in the Selected Tools

Tool/Reference	Compilation	Plagiarism	Code Quality	Software Metrics	Test Development	Funct. (Outcome)	Funct. (Source Code)	Custom Scripts
Web-CAT [75]	✓	✓	✓(Checksttyle)		✓(Coverage)	OC + UT		✓
PASS [255]	✓	✓				OC[1]		✓
Kattis [77]	✓	✓				OC		
ECSpooler [13]	✓					OC + UT		
STAGE (CodeCover) [172]	✓				✓(Coverage)	UT		
JACK [30, 136, 227]	✓	✓		✓		OC		✓
URI Online Judge [32]	✓	✓				OC		✓
CodeAssessor [241]	✓					OC		
VPL [203]	✓	✓				OC		✓
CloudCoder [108]	✓					OC		
CodeRunner [60]	✓					OC		
JPLAS [83–85]	✓					UT		
LX [61]	✓					OC + UT		
Testovid within Protus [242]	✓		✓(Checkstyle)	✓		UT		
TestMyCode [194]	✓		✓(Checkstyle)			UT		✓
CodeWorkout [74]	✓					UT		
Arena [184]	✓					UT		
JavAssess (ASys) [115]	✓			JR		OC	JR	✓
CodeOcean [226]	✓					OC + UT		✓
Jutge.org [176]	✓			✓		OC[2]		✓
GradeIT (PrUTOR) [62, 173]	✓(APR - Syntax)					OC		✓
neoESPA [148, 150]	✓	✓	✓(SonarQube)			OC		
Enki (Mooshak) [170]	✓					OC		✓
ProgEdu [50, 252]	✓	✓	✓			UT		✓
Aristotle [2]	✓					OC		✓
Submitty [180]	✓	✓	✓	✓	✓(Coverage)	OC + UT		✓
Codeflex [40]	✓					OC		
Dante [70]	✓	✓		✓		OC		✓
Coderiu [28]	✓					UT		✓
Edgar [156]	✓					OC		✓

[1]Token Pattern Approach.
[2]Several variants that are more tolerant.
JR, Java reflection; UT, unit testing; OC, output comparison.

Regarding feedback to the student, the selected tools have been surveyed according to the following features: failure info (information (or logs) about what failed); evaluation logs (the complete log of the evaluation); failed tests (the failing test or an associated message); how to proceed (either a fix or a hint on how to overcome an issue); report (a structured report of the evaluation including several details, i.e., execution times, resource usage, and tests passed and failed); and manual (messages written by the instructor about a submitted program). Table 6 overviews the feedback elements applied in the selected tools.

Finally, tools have been evaluated on mechanisms for data collection, including the presence of an embedded code editor and whether submissions, logging data (e.g., execution logs from submission execution), and code snapshots are stored, and learning reporting and analytics, including the generation of an evaluation report (i.e., a structured summary of the evaluation including, e.g., the classification, execution logs, test results, and execution time), visualization of simple statistics, and advanced analysis. Simple statistics comprises organizing, displaying, and summarizing data from server interactions (login activity, submissions, obtained grades, and classifications, etc.). In contrast, advanced analysis involves drawing conclusions about the data as well as collecting and presenting information to enable deeper insights about students' behavior during solution development (e.g., code progression). Table 7 summarizes the results.

6 DISCUSSION AND CONCLUSION

Automated assessment in CS has been continuously developing over the past decade. This review presented the most relevant developments of automated assessment for CS assignments, focusing on programming tasks. Considering the most recent systematic literature reviews, the notable advances reported in this review concentrate on five aspects: types of activities that can be automatically assessable, testing techniques, untrusted code execution approaches, feedback on how to overcome difficulties, and learning analytics from automated assessment data.

Table 5. Security Models for Untrusted Code Execution Used in the Selected Tools

Tool/Reference	Model	External
Web-CAT [75]	JVM (Java)	YES
PASS [255]	User permissions	YES
Kattis [77]	User permissions	YES
ECSpooler [13]	User permissions	YES
STAGE (CodeCover) [172]	JVM	YES
JACK [30, 136, 227]	JVM	YES
URI Online Judge [32]	User permissions	YES
CodeAssessor [241]	User permissions	NO
VPL [203]	Jail process	YES
CloudCoder [108]	JVM (Java) + Jail process	YES
CodeRunner [60]	User permissions	YES
JPLAS [83–85]	Blacklist + JVM	NO
LX [61]	User permissions	NO
Testovid within Protus [242]	JVM	YES
TestMyCode [194]	VM	YES
CodeWorkout [74]	JVM (Java) + User permissions	NO
Arena [184]	Container	YES
JavAssess (ASys) [115]	JVM	NO
CodeOcean [226]	Container	YES
Jutge.org [176]	Jail process	YES
GradeIT (PrUTOR) [62, 173]	Container	YES
neoESPA [148, 150]	User permissions	NO
Enki (Mooshak) [170]	User permissions	NO
ProgEdu [50, 252]	Container	YES
Aristotle [2]	User permissions	NO
Submitty [180]	Container	YES
Codeflex [40]	Jail process	YES
Dante [70]	User permissions	YES
Coderiu [28]	Container	YES
Edgar [156]	Container	YES

Researchers have made efforts to extend automated assessment to almost all practical tasks of the CS curriculum, which is not surprising, as the increasing number of enrollments and consequent higher instructors' workload demands automated assessment [56]. Moreover, the impact of the industry in CS curriculum and the approximation of enterprise-level automated testing with academic automated assessment has become more evident [41, 88, 91, 153, 214]. Even though feedback tends to be unsatisfactory, the robustness of industrial testing tools and the added benefit of familiarizing students with the tools they are likely going to use in the near future can be a compensating approach.

The past decade underwent a slight shift in what is considered to be a correct, partially correct, and incorrect solution by most practitioners. First, producing the expected output is not by itself the only factor that weighs in the grade of the program [27]. The adopted strategy, quality of the source code, algorithmic complexity, and many other aspects are also relevant indicators and thus may affect grades [27] and originate feedback to improve learning and promote good

Table 6. Feedback Elements Applied in the Selected Tools

Tool/Reference	Failure Info	Evaluation Logs	Failed Tests	How to Proceed	Report	Manual
Web-CAT [75]	✓		✓		✓	
PASS [255]	✓		✓			
Kattis [77]	✓				✓	
ECSpooler [13]	✓		✓			
STAGE (CodeCover) [172]	✓		✓		✓(Coverage)	
JACK [30, 136, 227]	✓		✓		✓	✓
URI Online Judge [32]	✓				✓	
CodeAssessor [241]	✓		✓			
VPL [203]	✓		✓			
CloudCoder [108]	✓		✓			
CodeRunner [60]	✓		✓			
JPLAS [83–85]	✓		✓			
LX [61]	✓		✓			
Testovid within Protus [242]	✓		✓			✓
TestMyCode [194]	✓		✓			✓
CodeWorkout [74]	✓		✓			
Arena [184]	✓	✓	✓			
JavAssess (ASys) [115]	✓		✓	✓	✓	✓
CodeOcean [226]	✓		✓			✓
Jutge.org [176]	✓		✓		✓	
GradeIT (PrUTOR) [62, 173]	✓		✓	✓[1]		
neoESPA [148, 150]	✓				✓(Quality)	
Enki (Mooshak) [170]	✓				✓	✓
ProgEdu [50, 252]	✓	✓	✓		✓(Quality)	
Aristotle [2]						
Submitty [180]	✓	✓	✓		✓	✓
Codeflex [40]	✓		✓			
Dante [70]	✓		✓		✓	✓
Coderiu [28]	✓	✓	✓		✓	
Edgar [156]	✓		✓		✓	

[1]Clustering with feedback rules + APR (syntax errors).

Table 7. Data Collection, Analytics, and Reporting Included in the Selected Tools

Tool/Reference	Data Collection				Data Analytics and Reporting		
	Editor	Submissions	Logging	Code Snapshots	Evaluation Report	Simple Statistics	Advanced Analysis
Web-CAT [75]		✓	✓		✓	✓	✓[1]
PASS [255]	✓	✓	✓		✓	✓	
Kattis [77]	✓	✓			✓	✓	
ECSpooler [13]		✓			✓		
STAGE (CodeCover) [172]		✓			✓	✓(Moodle)	
JACK [30, 136, 227]	✓	✓			✓	✓	
URI Online Judge [32]	✓	✓			✓	✓	
CodeAssessor [241]	✓	✓			✓		
VPL [203]	✓	✓			✓	✓(Moodle)	
CloudCoder [108]	✓	✓	✓	✓	✓	✓	
CodeRunner [60]	✓	✓			✓	✓(Moodle)	
JPLAS [83–85]	✓	✓			✓	✓	
LX [61]	✓	✓				✓(Moodle)	
Testovid within Protus [242]	✓	✓	✓		✓	✓	
TestMyCode [194]	✓	✓	✓	✓	✓	✓	
CodeWorkout [74]	✓	✓	✓		✓	✓	
Arena [184]	✓	✓	✓	✓	✓		
JavAssess (ASys) [115]		✓			✓		
CodeOcean [226]	✓	✓	✓		✓	✓	
Jutge.org [176]	✓	✓			✓	✓	
GradeIT (PrUTOR) [62, 173]	✓	✓	✓	✓	✓	✓	✓[2]
neoESPA [148, 150]		✓			✓		
Enki (Mooshak) [170]	✓	✓	✓		✓	✓	
ProgEdu [50, 252]		✓	✓	✓	✓	✓	
Aristotle [2]		✓			✓	✓	
Submitty [180]		✓	✓		✓	✓	
Codeflex [40]	✓	✓			✓	✓	
Dante [70]	✓	✓	✓		✓	✓	
Coderiu [28]	✓	✓	✓		✓	✓	
Edgar [156]	✓	✓	✓		✓	✓	✓[3]

[1]Learning progress graphs.
[2]Code size variation, code save progression, syntactic analytics, compilation error timeline, execution sequence.
[3]Student behavior and question analytics.

practices [238]. Then, in the limit, a program with a syntax error might be closer to the solution than a program failing a single test case. For instance, a correct Java program only missing a semicolon would raise a syntax error during evaluation and be awarded an incorrect classification without even being subject to further analysis in traditional testing approaches. Nevertheless, it is arguably closer to the solution than a program that does not consider an edge case.

With respect to this and other similar situations, static analysis approaches are increasingly becoming the research trend in the topic of automated testing techniques [10, 115, 149, 160, 258], as they "look" directly at the source code or its abstract representations, providing a more fair evaluation. This assessment perspective enables, for instance, personalized feedback on how to proceed, as it offers a better understanding of the student's program, such as the followed strategy [15], committed errors [249], and similarity (or missing steps) to a solution [95, 100].

Another major novelty is the containerization for untrusted code execution in isolation. Even though one cannot guarantee complete isolation out-of-the-box as in common virtualization, the lower overhead and all of the benefits of virtualization (e.g., consistent runs, increased portability, and improved scalability) make it a tempting model even if some tinkering is required to achieve "full security." The rapid proliferation of this approach in the second half of the past decade [62, 156, 180, 226] clearly pinpoints the beginning of a transition phase to containerization. Considering the eras of Douce et al. [68], after the era of web-based tools, we would say that we are currently facing the "Era of Containerization."

The rising of Big Data also impacted automated assessment, as data collection and learning analytics are growing interest as features for tools providing automated assessment. As these tools become more capable and feature rich, they are also conquering their space in different phases of learning and being actively used throughout CS courses. Hence, the quantity and quality of the data they produce increases significantly, enabling the extraction of unprecedented knowledge about students, such as their behavioral patterns [36, 51] and learning flaws [59]. In this sense, efforts to standardize the collection, sharing, and analysis of such data are under way [193]. Notwithstanding, there is still a large amount of wasted data (e.g., even though most of the surveyed tools provide an embedded editor, only five capture code changes, of which only two capture at keystroke granularity).

Furthermore, the actual implementation of recent advancements in automated assessment tools is far from concluded. Typically, a prototype is developed, then tested with a group of students under specific constraints, and the results are reported in the literature. As the constraints of the experiment are hardly replicated in a real scenario, the prototypes are either not further developed or remain closed. Nevertheless, unlike the previous reviews [6, 112, 220], more than half of the selected tools are open-source, which is a long-expected milestone to intensify developments in the field.

Figure 3 presents a bubble chart with the number of works done during the past decade in each of the selected topics of development, namely (1) Tool (research works reporting the development of an automated assessment tool); (2) OSS (from (1), those that are open-source); (3) Containerization (from (1), those that use containers to run students' programs); (4) Static Analysis (publications describing assessment techniques, tools, reviews, or experiments capitalizing on static analyses); (5) Clustering (new developments using clustering techniques); (6) Feedback (publications mostly related with feedback); (7) APR (from (6), those relying on automatic program repair); (8) Hint Generation (from (6), those involving the generation of hints); (9) Personalization (developments on techniques that consider the users' activity/progress); (10) Learning Analytics (papers reporting learning analytics developments); and (11) Students' Behavior (from (11), those focusing on analyzing students' behavior). This confirms all of the previous conclusions, particularly regarding the continuous development of new tools and the growing number of open-source software

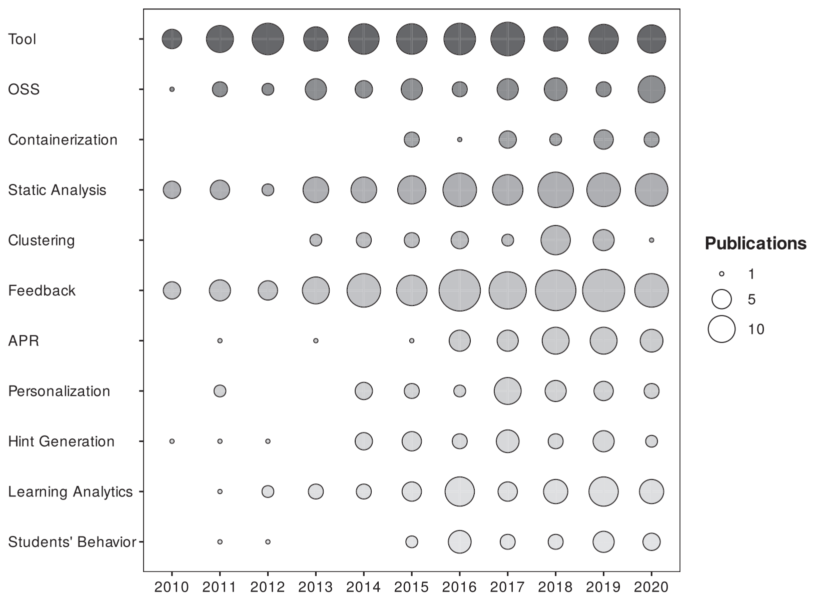

Fig. 3. Key topics of developments over the past decade.

among them. Moreover, the increasing interest in containerization of code execution, static analysis, clustering techniques, automatic program repair, generation of hints, personalization, learning analytics, and analyzing students' behavior is noticeable.

6.1 Threats to Validity

The scope and methodology adopted for this review poses several threats to its validity. The main concerns are the following.

Relevant tools missing. The conducted survey regarding tools could not cover each and every tool reported in the literature, as it would be both too extensive and time consuming. We have tried to select the 15 most relevant tools and then applied random sampling to pick another 15 tools. However, we acknowledge that some notable tools may have been left out.

Misinterpretation in data analysis. The considerable amount of effort involved in this review, and the consequent time it took to analyze the collected evidence, may have affected the interpretation of the main contributions of some studies. In particular, some papers present either multiple techniques for different phases of the assessment (e.g., testing and feedback) or backup techniques (e.g., if the reported method fails), which makes the process quite difficult to parse and somewhat subjective. Moreover, tools are typically only briefly described in the papers, without much detail. To address the former, one of the authors extracted the characteristics from all of the studies, and the other authors validated them after. Concerning the latter, we widened our investigation to related papers about the tools and explored the web to find additional references to the tool.

6.2 Educational Effectiveness (RQ6)

There are only a few empirical studies on the educational effectiveness of automated assessment in CS education. Nevertheless, there is sufficient research evidence to demonstrate its overall usefulness and educational benefits. In particular, it has been demonstrated that it (1) significantly reduces teachers' workload [77, 172, 177], (2) improves student learning [177], (3) increases the number of activities solved [70, 86], (4) is well-received by students [77, 86], and (5) engages students with activities without teachers' presence [87]. Moreover, some of these advantages are valid not only for formative assessment but also for summative assessment [44, 60]. However, several of these studies also indicate significant pedagogical gaps and deficiencies of automated assessment compared to human evaluation, especially regarding the provided feedback information, evaluation of partially correct programs, and critique of other code aspects [137, 143, 206]. This is in line with the problems being addressed in recent research trends, as discussed previously.

Therefore, one question intrinsic to any novel educational technique/tool must be "how effective is it?" Measuring the educational effectiveness of an automated assessment approach is not straightforward, as the multitude of approaches and aims adopted in the literature for such task indicate. For instance, a common approach to judge the effectiveness of an assessment technique is to compare offline (i.e., using previously collected data) its grading quality to that of a human grader [18, 27, 149, 173], whereas evaluating the effectiveness of tools and feedback techniques typically consists of analyzing the difference in students' attitudes/results with and without the "treatment" [13, 70, 143, 172]. The objective of the former is to approximate automated grading and feedback to that of a human grader, considering the feedback of teachers as optimal in terms of educational effectiveness. The latter compares the real effects of a new approach on the students.

From this perspective, there is some empirical evidence that endorses current research trends on static analysis approaches to solve aforementioned gaps. For instance, static analysis based techniques have proven success in grading assignments similarly to human graders [18, 149, 173] while being more consistent [173]. In fact, automatically generated feedback on how to proceed from techniques based on APR, semantic similarity, and symbolic execution is already promising (i.e., useful to struggling students) [95, 131, 254] and supports the need for further investigation to improve quality and time performance [95, 254]. Furthermore, these automated assessment techniques also seem to be capable of promoting and enforcing code quality [17, 150].

Unfortunately, not many empirical studies have been conducted to evaluate the educational effectiveness of developed works. Existing studies are mainly small experiments part of the papers introducing the novelty, to validate an early version of the proposal. This prevents driving research efforts with enough confidence and highlights the need for such studies.

6.3 Future Trends and Research Directions (RQ7)

The 2020 ITiCSE conference held online had a keynote on the history of CS education, by Matti Tedre [233]. It originated a question on how the keynote speaker expects programming teaching to be in a few decades. The answer was surprising: "Programming is a means to an end. The end is to make the machine perform the operations we want. We want to teach kids how to control the machines. In this perspective, programming, as we know, will not exist in the future." After having a moment of reflection, the signs of such evolution are present from the very beginning. The way we develop programs has been constantly evolving, from the time when a machine operator was needed for manually processing some parts of the program to the first programming languages that established the different language paradigms and the recent popularization of visual programming. Each time, we are adding another abstraction layer between the programmer and the machine.

There is certainly a long way to go until programming, as we know it, turns unnecessary to control a machine, if that ever happens. Nonetheless, it is of utmost importance to start prioritizing meaning over syntax, as the latter is ephemeral and, above all, dependent on the chosen programming language. This prioritization is also linked to judging how the student conceptualizes her solution rather than the mere output of her program. That is the idea beyond most static analysis approaches presented this past decade, but it also is a trend for the upcoming years, as there is still much room for improvement in the proposed techniques and novel approaches, such as reasoning on semantic graph representations of programs in large codebases to estimate the minimum number of changes to achieve a semantically correct solution.

Similar approaches can be applied in learning analytics to study students' behaviors, identify learning gaps, and predict failures early to make the teacher aware so that she can both act timely and deliver personalized feedback. The history of code modifications may give even better insights into how students think and approach the problem over submissions. Furthermore, it enables the analysis of students who either do not or rarely submit, whom the teachers would neglect since they have no information about their attempts. Therefore, some research lines will likely explore these opportunities, both for clusters of students and individuals. Figure 3 already provides a great insight into the trending personalization of automated assessment.

Following the recent efforts to standardize data produced from automated assessment [193], works adhering to the proposed format and even extending it are expected, such as adapting the format to store subsequent code states of the same program as changes in the AST (or other intermediate representation of source-code) or reasoning on previous collected data to suggest fixes to new students facing difficulties. Notwithstanding, it is important that such a format is widely adopted by automated assessment tools, as much data produced from automated assessment is currently wasted. Furthermore, it opens up the possibility to share not only datasets but also analysis tools, reducing the amount of work redone across different research groups.

Finally, there is a strong need for empirical studies examining the role and value of automated assessment techniques in the teaching and learning of CS (see Section 6.2). Most of the proposed works are only evaluated in a small and homogeneous group using an unfinished version of the tool, which is insufficient to establish its educational effectiveness. Hopefully, the next decade will also bring forward empirical research evidence on several techniques/tools.

Developments in the field of automated assessment are expected to maintain or even increase the current growth rate in upcoming years (or decades) as the demand for a highly skilled technical CS workforce continues its rise and online learning becomes even more part of the educational ecosystem [7]. Therefore, we recommend reviews similar to this and others valuable in the past [6, 68, 112] every 5 years. This suggestion is based on how much the automated assessment evolved between 2005 [6, 68] and 2010 [112], and the complexity of conducting a decade-long review similar to this.

REFERENCES

[1] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. 2010. Mutation analysis vs. code coverage in automated assessment of students' testing skills. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA'10)*. ACM, New York, NY, 153–160. https://doi.org/10.1145/1869542.1869567

[2] Michael D. Adams. 2017. Aristotle: A flexible open-source software toolkit for semi-automated marking of programming assignments. In *Proceedings of the 2017 IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM'17)*. IEEE, Los Alamitos, CA, 1–6. https://doi.org/10.1109/PACRIM.2017.8121888

[3] Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. 2006. Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In *Proceedings of the 2006 6th Baltic Sea Conference on Computing Education Research: Koli Calling (Baltic Sea'06)*. ACM, New York, NY, 141–142. https://doi.org/10.1145/1315803.1315831

[4] Alex Aiken. 2021. MOSS: A System for Detecting Software Similarity. Retrieved September 22, 2021 from https://theory.stanford.edu/~aiken/moss/.

[5] Kirsti Ala-Mutka, Toni Uimonen, and Hannu-Matti Jarvinen. 2004. Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education: Research* 3, 1 (Jan. 2004), 245–262. https://www.learntechlib.org/p/111452.

[6] Kirsti M. Ala-Mutka. 2005. A survey of automated assessment approaches for programming assignments. *Computer Science Education* 15, 2 (June 2005), 83–102. https://doi.org/10.1080/08993400500150747

[7] Hasan Alkhatib, Paolo Faraboschi, Eitan Frachtenberg, Hironori Kasahara, Danny Lange, Phil Laplante, Arif Merchant, Dejan Milojicic, and Karsten Schwan. 2021. IEEE Computer Society 2022 Report. Retrieved September 22, 2021 from https://www.computer.org/publications/tech-news/trends/2022-report.

[8] Anthony Allevato and Stephen H. Edwards. 2012. RoboLIFT: Engaging CS2 students with testable, automatically evaluated Android applications. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE'12)*. ACM, New York, NY, 547–552. https://doi.org/10.1145/2157136.2157293

[9] Anthony Allowatt and Stephen H. Edwards. 2005. IDE support for test-driven development and automated grading in both Java and C++. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange (eclipse'05)*. ACM, New York, NY, 100–104. https://doi.org/10.1145/1117696.1117717

[10] F. AlShamsi and A. Elnagar. 2011. An automated assessment and reporting tool for introductory Java programs. In *Proceedings of the 2011 International Conference on Innovations in Information Technology*. IEEE, Los Alamitos, CA, 324–329. https://doi.org/10.1109/INNOVATIONS.2011.5893842

[11] Rajeev Alur, Loris D'Antoni, Sumit Gulwani, Dileep Kini, and Mahesh Viswanathan. 2013. Automated grading of DFA constructions. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13)*. 1976–1982. https://doi.org/10.5555/2540128.2540412

[12] Mario Amelung, Peter Forbrig, and Dietmar Rösner. 2008. Towards generic and flexible web services for e-assessment. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'08)*. ACM, New York, NY, 219–224. https://doi.org/10.1145/1384271.1384330

[13] Mario Amelung, Katrin Krieger, and Dietmar Rosner. 2011. E-assessment as a service. *IEEE Transactions on Learning Technologies* 4, 2 (April 2011), 162–174. https://doi.org/10.1109/TLT.2010.24

[14] V. Anjali, T. R. Swapna, and Bharat Jayaraman. 2015. Plagiarism detection for Java programs without source codes. *Procedia Computer Science* 46 (2015), 749–758. https://doi.org/10.1016/j.procs.2015.02.143

[15] Paolo Antonucci, Christian Estler, Durica Nikolic, Marco Piccioni, and Bertrand Meyer. 2015. An incremental hint system for automated programming assignments. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, 320–325. https://doi.org/10.1145/2729094.2742607

[16] Kazuki Anzai and Yutaka Watanobe. 2019. Algorithm to determine extended edit distance between program codes. In *Proceedings of the 2019 IEEE 13th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip (MCSoC'19)*. IEEE, Los Alamitos, CA, 180–186. https://doi.org/10.1109/MCSoC.2019.00033

[17] Eliane Araujo, Dalton Serey, and Jorge Figueiredo. 2016. Qualitative aspects of students' programs: Can we make them measurable? In *Proceedings of the 2016 IEEE Frontiers in Education Conference (FIE'16)*. IEEE, Los Alamitos, CA, 1–8. https://doi.org/10.1109/FIE.2016.7757725

[18] Sara Mernissi Arifi, Azeddine Zahi, and Rachid Benabbou. 2016. Semantic similarity based evaluation for C programs through the use of symbolic execution. In *Proceedings of the 2016 IEEE Global Engineering Education Conference (EDUCON'16)*. IEEE, Los Alamitos, CA, 826–833. https://doi.org/10.1109/EDUCON.2016.7474648

[19] Tapio Auvinen. 2015. Harmful study habits in online learning environments with automatic assessment. In *Proceedings of the 2015 International Conference on Learning and Teaching in Computing and Engineering*. IEEE, Los Alamitos, CA, 50–57. https://doi.org/10.1109/latice.2015.31

[20] Maha Aziz, Heng Chi, Anant Tibrewal, Max Grossman, and Vivek Sarkar. 2015. Auto-grading for parallel programs. In *Proceedings of the Workshop on Education for High-Performance Computing (EduHPC'15)*. ACM, New York, NY, 1–8. https://doi.org/10.1145/2831425.2831427

[21] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys* 51, 3 (May 2018), Article 50, 39 pages. https://doi.org/10.1145/3182657

[22] E. F. Barbosa, J. C. Maldonado, R. LeBlanc, and M. Guzdial. 2003. Introducing testing practices into objects and design course. In *Proceedings of the 2003 16th Conference on Software Engineering Education and Training, (CSEE&T '03)*. IEEE, Los Alamitos, CA, 279–286. https://doi.org/10.1109/CSEE.2003.1191387

[23] Enrique Flores, Alberto Barron-Cedeno, Lidia Moreno, and Paolo Rosso. 2015. Cross-language source code re-use detection using latent semantic analysis. *Journal of Universal Computer Science* 21, 13 (Dec. 2015), 1708–1725. https://doi.org/10.3217/JUCS-021-13-1708

[24] Lewis Baumstark and Edwin Rudolph. 2013. Automated online grading for virtual machine-based systems administration courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13)*. ACM, New York, NY, 477. https://doi.org/10.1145/2445196.2445340

[25] Kent Beck. 1994. Simple smalltalk testing: With patterns. *Smalltalk Report* 4, 2 (1994), 16–18.

[26] Andrés M. Bejarano, Lucy E. García, and Eduardo E. Zurek. 2015. Detection of source code similitude in academic environments. *Computer Applications in Engineering Education* 23, 1 (2015), 13–22. https://doi.org/10.1002/cae.21571 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.21571

[27] Clara Benac Earle, Lars-Ake Fredlund, and John Hughes. 2016. Automatic grading of programming exercises using property-based testing. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, 47–52. https://doi.org/10.1145/2899415.2899443

[28] Guido Benetti, Gianluca Roveda, Davide Giuffrida, and Tullio Facchinetti. 2019. Coderiu: A cloud platform for computer programming e-learning. In *Proceedings of the 2019 IEEE 17th International Conference on Industrial Informatics (INDIN'19)*, Vol. 1. IEEE, Los Alamitos, CA, 1126–1132. https://doi.org/10.1109/INDIN41052.2019.8972058

[29] S. D. Benford, E. K. Burke, E. Foxley, and C. A. Higgins. 1995. The Ceilidh system for the automatic grading of students on programming courses. In *Proceedings of the 33rd Annual Southeast Regional Conference (ACM-SE'95)*. ACM, New York, NY, 176–182. https://doi.org/10.1145/1122018.1122050

[30] Marc Berges, Michael Striewe, Philipp Shah, Michael Goedicke, and Peter Hubwieser. 2016. Towards deriving programming competencies from student errors. In *Proceedings of the 2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE'16)*. IEEE, Los Alamitos, CA, 19–23. https://doi.org/10.1109/LaTiCE.2016.6

[31] R. E. Berry. 1966. Grader programs. *Computer Journal* 9, 3 (11 1966), 252–256. https://doi.org/10.1093/comjnl/9.3.252

[32] Jean Luca Bez, Neilor A. Tonin, and Paulo R. Rodegheri. 2014. URI online judge academic: A tool for algorithms and programming classes. In *Proceedings of the 2014 9th International Conference on Computer Science and Education*. IEEE, Los Alamitos, CA, 149–152. https://doi.org/10.1109/ICCSE.2014.6926445

[33] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic program corrector for introductory programming assignments. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, New York, NY, 60–70. https://doi.org/10.1145/3180155.3180219

[34] Weiyi Bian, Omar Alam, and Jörg Kienzle. 2020. Is automated grading of models effective? Assessing automated grading of class diagrams. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS'20)*. ACM, New York, NY, 365–376. https://doi.org/10.1145/3365438.3410944

[35] John Biggs and Catherine Tang. 2011. *Teaching for Quality Learning at University: What the Student Does?* (4th. ed.). Maidenhead, Berkshire, UK.

[36] Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller. 2014. Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences* 23, 4 (2014), 561–599. https://doi.org/10.1080/10508406.2014.954750

[37] Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana Franklin. 2013. Hairball: Lint-inspired static analysis of scratch projects. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13)*. ACM, New York, NY, 215–220. https://doi.org/10.1145/2445196.2445265

[38] Younes Boubekeur, Gunter Mussbacher, and Shane McIntosh. 2020. Automatic assessment of students' software models using a simple heuristic and machine learning. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS'20)*. ACM, New York, NY, Article 20, 10 pages. https://doi.org/10.1145/3417990.3418741

[39] David Boud and Elizabeth Molloy (Eds.). 2013. *Feedback in Higher and Professional Education: Understanding It and Doing It Well.* Routledge, London, UK.

[40] Miguel Brito and Celestino Goncalves. 2019. Codeflex: A web-based platform for competitive programming. In *Proceedings of the 2019 14th Iberian Conference on Information Systems and Technologies (CISTI'19)*. IEEE, Los Alamitos, CA, 1–6. https://doi.org/10.23919/CISTI.2019.8760776

[41] Daniel Bruzual, Maria L. Montoya Freire, and Mario Di Francesco. 2020. Automated assessment of Android exercises with cloud-native technologies. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, 40–46. https://doi.org/10.1145/3341525.3387430

[42] Utharn Buranasaksee. 2017. LINSIM: A framework of an automatic assessment for Linux-based operating system exercises. In *Proceedings of the 5th International Conference on Information and Education Technology (ICIET'17)*. ACM, New York, NY, 121–125. https://doi.org/10.1145/3029387.3029400

[43] Barry Burd, João Paulo Barros, Chris Johnson, Stan Kurkovsky, Arnold Rosenbloom, and Nikolai Tillman. 2012. Educating for mobile computing: Addressing the new challenges. In *Proceedings of the Final Reports on Innovation and Technology in Computer Science Education 2012 Working Groups (ITiCSE-WGR'12)*. ACM, New York, NY, 51–63. https://doi.org/10.1145/2426636.2426641

[44] Yingjun Cao, Leo Porter, Soohyun Nam Liao, and Rick Ord. 2019. Paper or Online? A comparison of exam grading techniques. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, 99–104. https://doi.org/10.1145/3304221.3319739

[45] Dong-Kyu Chae, Jiwoon Ha, Sang-Wook Kim, BooJoong Kang, and Eul Gyu Im. 2013. Software plagiarism detection: A graph-based approach. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management (CIKM'13)*. ACM, New York, NY, 1577–1580. https://doi.org/10.1145/2505515.2507848

[46] Pinaki Chakraborty, P. C. Saxena, and C. P. Katti. 2011. Fifty years of automata simulation: A review. *ACM Inroads* 2, 4 (Dec. 2011), 59–70. https://doi.org/10.1145/2038876.2038893

[47] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. 2003. On automated grading of programming assignments in an academic institution. *Computers & Education* 41, 2 (2003), 121–131. https://doi.org/10.1016/S0360-1315(03)00030-7

[48] Hayden Cheers, Yuqing Lin, and Shamus P. Smith. 2021. Academic source code plagiarism detection by measuring program behavioral similarity. *IEEE Access* 9 (2021), 50391–50412. https://doi.org/10.1109/ACCESS.2021.3069367

[49] Di Chen, Hui Li, Mei Chen, Zhenyu Dai, Huanjun Li, Ming Zhu, and Jian Zhang. 2020. PeBAO: A performance bottleneck analysis and optimization framework in concurrent environments. In *Intelligent Algorithms in Software Engineering*, Radek Silhavy (Ed.). Springer International, Cham, Switzerland, 248–260.

[50] Hsi-Min Chen, Wei-Han Chen, Nien-Lin Hsueh, Chi-Chen Lee, and Chia-Hsiu Li. 2017. ProgEdu—An automatic assessment platform for programming courses. In *Proceedings of the 2017 International Conference on Applied System Innovation (ICASI'17)*. IEEE, Los Alamitos, CA, 173–176. https://doi.org/10.1109/ICASI.2017.7988376

[51] H. M. Chen, B. A. Nguyen, Y. X. Yan, and C. R. Dow. 2020. Analysis of learning behavior in an automated programming assessment environment: A code quality perspective. *IEEE Access* 8 (2020), 167341–167354. https://doi.org/10.1109/ACCESS.2020.3024102

[52] Rong Chen, Lina Hong, Chunyan Lu, and Wu Deng. 2010. Author identification of software source code with program dependence graphs. In *Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops (COMPSACW'10)*. IEEE, Los Alamitos, CA, 281–286. https://doi.org/10.1109/COMPSACW.2010.56

[53] Sammi Chow, Kalina Yacef, Irena Koprinska, and James Curran. 2017. Automated data-driven hints for computer programming students. In *Adjunct Publication of the 25th Conference on User Modeling, Adaptation, and Personalization (UMAP'17)*. ACM, New York, NY, 5–10. https://doi.org/10.1145/3099023.3099065

[54] Benjamin Clegg, Siobhan North, Phil McMinn, and Gordon Fraser. 2019. Simulating student mistakes to evaluate the fairness of automated grading. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET'19)*. IEEE, Los Alamitos, CA, 121–125. https://doi.org/10.1109/ICSE-SEET.2019.00021

[55] Sabastien Combefis and Arnaud Schils. 2016. Automatic programming error class identification with code plagiarism-based clustering. In *Proceedings of the 2nd International Code Hunt Workshop on Educational Software Engineering*. ACM, New York, NY, 1–6. https://doi.org/10.1145/2993270.2993271

[56] Computing Research Association. 2017. *Generation CS: Computer Science Undergraduate Enrollments Surge since 2006*. Computing Research Association. https://cra.org/data/Generation-CS/.

[57] Helder Correia, José Paulo Leal, and José Carlos Paiva. 2018. Improving diagram assessment in Mooshak. In *Technology Enhanced Assessment*, Eric Ras and Ana Elena Guerrero Roldán (Eds.). Springer International, Cham, Switzerland, 69–82. https://doi.org/10.1007/978-3-319-97807-9_6

[58] Georgina Cosma and Mike Joy. 2012. An approach to source-code plagiarism detection and investigation using latent semantic analysis. *IEEE Transactions on Computers* 61, 3 (March 2012), 379–394. https://doi.org/10.1109/TC.2011.223

[59] Evandro B. Costa, Baldoino Fonseca, Marcelo Almeida Santana, Fabrisia Ferreira de Araujo, and Joilson Rego. 2017. Evaluating the effectiveness of educational data mining techniques for early prediction of students' academic failure in introductory programming courses. *Computers in Human Behavior* 73 (2017), 247–256. https://doi.org/10.1016/j.chb.2017.01.047

[60] David Croft and Matthew England. 2020. Computing with CodeRunner at Coventry University: Automated summative assessment of Python and C++ code. In *Proceedings of the 2020 4th Conference on Computing Education Practice*. ACM, New York, NY, 1–4. https://doi.org/10.1145/3372356.3372357

[61] Karol Danutama and Inggriani Liem. 2013. Scalable autograder and LMS integration. *Procedia Technology* 11 (2013), 388–395. https://doi.org/10.1016/j.protcy.2013.12.207

[62] Rajdeep Das, Umair Z. Ahmed, Amey Karkare, and Sumit Gulwani. 2016. Prutor: A system for tutoring CS1 and collecting student programs for analysis. *arXiv:1608.03828* [cs.CY] (2016).

[63] L. de Oliveira Brandao, Y. Bosse, and M. A. Gerosa. 2016. Visual programming and automatic evaluation of exercises: An experience with a STEM course. In *Proceedings of the 2016 IEEE Frontiers in Education Conference (FIE'16)*. IEEE, Los Alamitos, CA, 1–9. https://doi.org/10.1109/FIE.2016.7757621

[64] Michael de Raadt, Stijn Dekeyser, and Tien Yu Lee. 2006. Do students *SQLify*? Improving learning outcomes with peer review and enhanced computer assisted assessment of querying skills. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006 (Baltic Sea'06)*. ACM, New York, NY, 101–108. https://doi.org/10.1145/1315803.1315821

[65] Draylson M. de Souza, Bruno H. Oliveira, Jose Carlos Maldonado, Simone R. S. Souza, and Ellen F. Barbosa. 2014. Towards the use of an automatic assessment system in the teaching of software testing. In *Proceedings of the 2014 IEEE Frontiers in Education Conference (FIE'14)*. IEEE, Los Alamitos, CA, 1–8. https://doi.org/10.1109/FIE.2014.7044375

[66] Kyle Dewey, Phillip Conrad, Michelle Craig, and Elena Morozova. 2017. Evaluating test suite effectiveness and assessing student code via constraint logic programming. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'17).* ACM, New York, NY, 317–322. https://doi.org/10.1145/3059009.3059051

[67] Filip Dochy. 2006. A guide for writing scholarly articles or reviews for the educational research review. *Educational Research Review* 4, 1–2 (2006), 1–21.

[68] Christopher Douce, David Livingstone, and James Orwell. 2005. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing* 5, 3 (Sept. 2005), 4. https://doi.org/10.1145/1163405.1163409

[69] Xuliang Duan, Mantao Wang, and Jiong Mu. 2017. A plagiarism detection algorithm based on extended winnowing. *MATEC Web of Conferences* 128 (2017), 02019. https://doi.org/10.1051/matecconf/201712802019

[70] Piotr Duch and Tomasz Jaworski. 2018. Dante—Automated assessments tool for students' programming assignments. In *Proceedings of the 2018 11th International Conference on Human System Interaction (HSI'18).* IEEE, Los Alamitos, CA, 162–168. https://doi.org/10.1109/HSI.2018.8431146

[71] Tore Dyba and Torgeir Dingsoyr. 2008. Empirical studies of agile software development: A systematic review. *Information and Software Technology* 50, 9 (2008), 833–859. https://doi.org/10.1016/j.infsof.2008.01.006

[72] Stephen H. Edwards. 2003. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing* 3, 3 (Sept. 2003), 1–es. https://doi.org/10.1145/1029994.1029995

[73] Stephen H. Edwards. 2004. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'04).* ACM, New York, NY, 26–30. https://doi.org/10.1145/971300.971312

[74] Stephen H. Edwards and Krishnan Panamalai Murali. 2017. CodeWorkout: Short programming exercises with built-in data collection. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'17).* ACM, New York, NY, 188–193. https://doi.org/10.1145/3059009.3059055

[75] Stephen H. Edwards and Manuel A. Perez-Quinones. 2008. Web-CAT: Automatically grading programming assignments. In *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'08).* ACM, New York, NY, 328. https://doi.org/10.1145/1384271.1384371

[76] John English. 2004. Automated assessment of GUI programs using JEWL. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'04).* ACM, New York, NY, 137–141. https://doi.org/10.1145/1007996.1008033

[77] E. Enstrom, G. Kreitz, F. Niemela, P. Soderman, and V. Kann. 2011. Five years with Kattis—Using an automated assessment system in teaching. In *Proceedings of the 2011 Frontiers in Education Conference (FIE'11).* IEEE, Los Alamitos, CA, T3J-1–T3J-6. https://doi.org/10.1109/FIE.2011.6142931

[78] Michal Forišek. 2006. Security of programming contest systems. In *Information Technologies at School*, Valentina Dagiene and Roland Mittermeir (Eds.). 2nd International Conference on Informatics in Secondary Schools: Evolution and Perspectives, Vilnius, Lithuania, 553–563.

[79] George E. Forsythe and Niklaus Wirth. 1965. Automatic grading programs. *Communications of the ACM* 8, 5 (May 1965), 275–278. https://doi.org/10.1145/364914.364937

[80] J. S. Foundation. 2020. Appium: Mobile app automation made awesome. *JS Foundation.* Retrieved August 31, 2021 from http://appium.io/.

[81] Deqiang Fu, Yanyan Xu, Haoran Yu, and Boyang Yang. 2017. WASTK: A weighted abstract syntax tree kernel method for source code plagiarism detection. *Scientific Programming* 2017 (2017), 1–8. https://doi.org/10.1155/2017/7809047

[82] Xiang Fu, Boris Peltsverger, Kai Qian, Lixin Tao, and Jigang Liu. 2008. APOGEE: Automated project grading and instant feedback system for web based computing. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'08).* ACM, New York, NY, 77–81. https://doi.org/10.1145/1352135.1352163

[83] Nobuo Funabiki, Yukiko Matsushima, Toru Nakanishi, Kan Watanabe, and Noriki Amano. 2013. A Java programming learning assistant system using test-driven development method. *IAENG International Journal of Computer Science* 40, 1 (2013), 38–46.

[84] N. Funabiki, Y. Wang, N. Ishihara, and W. Kao. 2017. An offline answering function for code writing problem in Java programming learning assistant system. In *Proceedings of the 2017 IEEE International Conference on Consumer Electronics—Taiwan (ICCE-TW'17).* IEEE, Los Alamitos, CA, 241–242. https://doi.org/10.1109/ICCE-China.2017.7991085

[85] Nobuo Funabiki, Masaki Yamaguchi, Minoru Kuribayashi, Htoo Htoo Sandi Kyaw, Su Sandy Wint, Soe Thandar Aung, and Wen-Chung Kao. 2020. An extension of code correction problem for Java programming learning assistant system. In *Proceedings of the 2020 8th International Conference on Information and Education Technology.* ACM, New York, NY, 110–115. https://doi.org/10.1145/3395245.3396439

[86] Tommy Farnqvist and Fredrik Heintz. 2016. Competition and feedback through automated assessment in a data structures and algorithms course. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education.* ACM, New York, NY, 130–135. https://doi.org/10.1145/2899415.2899454

[87] Daniel Galan, Ruben Heradio, Hector Vargas, Ismael Abad, and Jose A. Cerrada. 2019. Automated assessment of computer programming practices: The 8-years UNED experience. *IEEE Access* 7 (2019), 130113–130119. https://doi.org/10.1109/ACCESS.2019.2938391

[88] Jianxiong Gao, Bei Pang, and Steven S. Lumetta. 2016. Automated feedback framework for introductory programming courses. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, 53–58. https://doi.org/10.1145/2899415.2899440

[89] Alex Gerdes, Johan T. Jeuring, and Bastiaan J. Heeren. 2010. Using strategies for assessment of programming exercises. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE'10)*. ACM, New York, NY, 441. https://doi.org/10.1145/1734263.1734412

[90] A. Gomes and A. Mendes. 2014. A teacher's view about introductory programming teaching and learning: Difficulties, strategies and motivations. In *Proceedings of the 2014 IEEE Frontiers in Education Conference (FIE'14)*. IEEE, Los Alamitos, CA, 1–8. https://doi.org/10.1109/FIE.2014.7044086

[91] Aldo Gordillo. 2019. Effect of an instructor-centered tool for automatic assessment of programming assignments on students' perceptions and performance. *Sustainability* 11, 20 (Oct. 2019), 5568. https://doi.org/10.3390/su11205568

[92] Eric Gramond and Susan H. Rodger. 1999. Using JFLAP to interact with theorems in automata theory. In *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'99)*. ACM, New York, NY, 336–340. https://doi.org/10.1145/299649.299800

[93] Geoffrey R. Gray and Colin A. Higgins. 2006. An introspective approach to marking graphical user interfaces. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE'06)*. ACM, New York, NY, 43–47. https://doi.org/10.1145/1140124.1140139

[94] F. Grivokostopoulou, I. Perikos, and I. Hatzilygeroudis. 2012. An automatic marking system for FOL to CF conversions. In *Proceedings of the IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE'12) 2012*. IEEE, Los Alamitos, CA, H1A-7–H1A-12. https://doi.org/10.1109/TALE.2012.6360317

[95] Sumit Gulwani, Ivan Readicek, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, 465–480. https://doi.org/10.1145/3192366.3192387

[96] Lasse Hakulinen and Lauri Malmi. 2014. QR code programming tasks with automated assessment. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education (ITiCSE'14)*. ACM, New York, NY, 177–182. https://doi.org/10.1145/2591708.2591761

[97] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D. Nguyen. 2018. Providing meaningful feedback for autograding of programming assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE'18)*. ACM, New York, NY, 278–283. https://doi.org/10.1145/3159450.3159502

[98] Maurice Howard Halstead. 1977. *Elements of Software Science*. Computer Science library, Vol. 7. Elsevier, New York, NY.

[99] John Hattie and Helen Timperley. 2007. The power of feedback. *Review of Educational Research* 77, 1 (2007), 81–112. https://doi.org/10.3102/003465430298487

[100] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Bjorn Hartmann. 2017. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the 2017 4th ACM Conference on Learning @ Scale*. ACM, New York, NY, 89–98. https://doi.org/10.1145/3051457.3051467

[101] Sarah Heckman and Jason King. 2018. Developing software engineering skills using real tools for automated grading. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, 794–799. https://doi.org/10.1145/3159450.3159595

[102] Bastiaan Heeren and Johan Jeuring. 2009. Recognizing strategies. *Electronic Notes in Theoretical Computer Science* 237 (2009), 91–106. https://doi.org/10.1016/j.entcs.2009.03.037

[103] J. B. Hext and J. W. Winings. 1969. An automatic grading scheme for simple programming exercises. *Communications of the ACM* 12, 5 (May 1969), 272–275. https://doi.org/10.1145/362946.362981

[104] Colin Higgins, Tarek Hegazy, Pavlos Symeonidis, and Athanasios Tsintsifas. 2003. The coursemarker CBA system: Improvements over Ceilidh. *Education and Information Technologies* 8, 3 (Sept. 2003), 287–304. https://doi.org/10.1023/A:1026364126982

[105] C. Higgins, P. Symeonidis, and A. Tsintsifas. 2002. Diagram-based CBA using DATsys and CourseMaster. In *Proceedings of the 2002 International Conference on Computers in Education*, Vol. 1. IEEE, Los Alamitos, CA, 167–172. https://doi.org/10.1109/CIE.2002.1185893

[106] Blake Hodgkinson, Christof Lutteroth, and Burkhard Wunsche. 2016. glGetFeedback—Toward automatic feedback and assessment for OpenGL 3D modelling assignments. In *Proceedings of the 2016 International Conference on Image and Vision Computing New Zealand (IVCNZ'16)*. IEEE, Los Alamitos, CA, 1–6. https://doi.org/10.1109/IVCNZ.2016.7804418

[107] Jack Hollingsworth. 1960. Automatic graders for programming classes. *Communications of the ACM* 3, 10 (Oct. 1960), 528–529. https://doi.org/10.1145/367415.367422

[108] David Hovemeyer and Jaime Spacco. 2013. CloudCoder: A web-based programming exercise system. *Journal of Computing Sciences in Colleges* 28, 3 (Jan. 2013), 30. https://doi.org/10.5555/2400161.2400167

[109] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-factoring based program repair applied to programming assignments. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. IEEE, Los Alamitos, CA, 388–398. https://doi.org/10.1109/ASE.2019.00044

[110] Christian Hundt, Moritz Schlarb, and Bertil Schmidt. 2017. SAUCE: A web application for interactive teaching and learning of parallel programming. *Journal of Parallel and Distributed Computing* 105 (July 2017), 163–173. https://doi.org/10.1016/j.jpdc.2016.12.028

[111] W. Hwang, C. Wang, G. Hwang, Y. Huang, and S. Huang. 2008. A web-based programming learning environment to support cognitive development. *Interacting with Computers* 20, 6 (2008), 524–534. https://doi.org/10.1016/j.intcom.2008.07.002

[112] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppala. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling'10)*. ACM, New York, NY, 86–93. https://doi.org/10.1145/1930464.1930480

[113] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H. Edwards, Essi Isohanni, et al. 2015. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports (ITICSE-WGR'15)*. ACM, New York, NY, 41–63. https://doi.org/10.1145/2858796.2858798

[114] Ushio Inoue and Shuhei Wada. 2012. Detecting plagiarisms in elementary programming courses. In *Proceedings of the 2012 9th International Conference on Fuzzy Systems and Knowledge Discovery*. IEEE, Los Alamitos, CA, 2308–2312. https://doi.org/10.1109/FSKD.2012.6234186

[115] David Insa and Josep Silva. 2018. Automatic assessment of Java code. *Computer Languages, Systems & Structures* 53 (Sept. 2018), 59–72. https://doi.org/10.1016/j.cl.2018.01.004

[116] Peter C. Isaacson and Terry A. Scott. 1989. Automating the execution of student programs. *SIGCSE Bulletin* 21, 2 (June 1989), 15–22. https://doi.org/10.1145/65738.65741

[117] Roman Ivanov. 2020. *Checkstyle*. SourceForge. Retrieved August 31, 2021 from https://checkstyle.sourceforge.io/.

[118] David Jackson and Michelle Usher. 1997. Grading student programs using ASSYST. In *Proceedings of the T28th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'97)*. ACM, New York, NY, 335–339. https://doi.org/10.1145/268084.268210

[119] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, Los Alamitos, CA, 215–224. https://doi.org/10.1145/1806799.1806833

[120] Yoon-Chan Jhi, Xinran Wang, Xiaoqi Jia, Sencun Zhu, Peng Liu, and Dinghao Wu. 2011. Value-based program characterization and its application to software plagiarism detection. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, NY, 756–765. https://doi.org/10.1145/1985793.1985899

[121] Wei Jin. 2020. Automatic grading for program tracing exercises. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, 1409–1409. https://doi.org/10.1145/3328778.3372561

[122] David E. Johnson. 2016. ITCH: Individual testing of computer homework for scratch assignments. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE'16)*. ACM, New York, NY, 223–227. https://doi.org/10.1145/2839509.2844600

[123] Mike Joy, Nathan Griffiths, and Russell Boyatt. 2005. The boss online submission and assessment system. *Journal on Educational Resources in Computing* 5, 3 (Sept. 2005), 2–es. https://doi.org/10.1145/1163405.1163407

[124] Mike Joy and Michael Luck. 1998. Effective electronic marking for on-line assessment. In *Proceedings of the 6th Annual Conference on the Teaching of Computing and the 3rd Annual Conference on Integrating Technology into Computer Science Education: Changing the Delivery of Computer Science Education (ITiCSE'98)*. ACM, New York, NY, 134–138. https://doi.org/10.1145/282991.283096

[125] Oscar Karnalim. 2017. An abstract method linearization for detecting source code plagiarism in object-oriented environment. In *Proceedings of the 2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS'17)*. IEEE, Los Alamitos, CA, 58–61. https://doi.org/10.1109/ICSESS.2017.8342863

[126] H. Ke, G. Zhang, and H. Yan. 2009. Automatic grading system on SQL programming. In *Proceedings of the 2009 International Conference on Scalable Computing and Communications and the 8th International Conference on Embedded Computing*. IEEE, Los Alamitos, CA, 537–540. https://doi.org/10.1109/EmbeddedCom-ScalCom.2009.105

[127] Caitlin Kelleher and Randy Pausch. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys* 37, 2 (June 2005), 83–137. https://doi.org/10.1145/1089733.1089734

[128] Wouter Kerdijk, René A. Tio, B. Florentine Mulder, and Janke Cohen-Schotanus. 2013. Cumulative assessment: Strategic choices to influence students' study effort. *BMC Medical Education* 13, 1 (Dec. 2013), 172. https://doi.org/10.1186/1472-6920-13-172

[129] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2014. Strategy-based feedback in a programming tutor. In *Proceedings of the Computer Science Education Research Conference (CSERC'14)*. ACM, New York, NY, 43–54. https://doi.org/10.1145/2691352.2691356

[130] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2019. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education* 19, 1 (Jan. 2019), 1–43. https://doi.org/10.1145/3231711

[131] Dohyeong Kim, Yonghwi Kwon, Peng Liu, I. Luk Kim, David Mitchel Perry, Xiangyu Zhang, and Gustavo Rodriguez-Rivera. 2016. Apex: Automatic programming assignment error explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, 311–327. https://doi.org/10.1145/2983990.2984031

[132] Diana Kirk, Tyne Crow, Andrew Luxton-Reilly, and Ewan Tempero. 2020. On assuring learning about code quality. In *Proceedings of the 22nd Australasian Computing Education Conference (ACE'20)*. ACM, New York, NY, 86–94. https://doi.org/10.1145/3373165.3373175

[133] Carsten Kleiner, Christopher Tebbe, and Felix Heine. 2013. Automated grading and tutoring of SQL statements to improve student learning. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research (Koli Calling'13)*. ACM, New York, NY, 161–168. https://doi.org/10.1145/2526968.2526986

[134] Matthias Kramer, Mike Barkmin, and Torsten Brinda. 2018. Evaluating responses in source code highlighting tasks: Preliminary considerations for automatic assessment. In *Proceedings of the 13th Workshop in Primary and Secondary Computing Education*. ACM, New York, NY, 1–4. https://doi.org/10.1145/3265757.3265775

[135] Matthias Kramer, Mike Barkmin, Torsten Brinda, and David Tobinski. 2018. Automatic assessment of source code highlighting tasks: Investigation of different means of measurement. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research (Koli Calling'18)*. ACM, New York, NY, 1–10. https://doi.org/10.1145/3279720.3279729

[136] Johannes Krugel, Peter Hubwieser, Michael Goedicke, Michael Striewe, Mike Talbot, Christoph Olbricht, Melanie Schypula, and Simon Zettler. 2020. Automated measurement of competencies and generation of feedback in object-oriented programming courses. In *Proceedings of the 2020 IEEE Global Engineering Education Conference (EDUCON'20)*. IEEE, Los Alamitos, CA, 329–338. https://doi.org/10.1109/EDUCON45650.2020.9125323

[137] Angelo Kyrilov and David C. Noelle. 2016. Do students need detailed feedback on programming exercises and can automated assessment systems provide it? *Journal of Computing Sciences in Colleges* 31, 4 (April 2016), 115–121. https://doi.org/10.5555/2904127.2904147

[138] Fabrizio Lamberti, Andrea Sanna, Gianluca Paravati, and Gilles Carlevaris. 2014. Automatic grading of 3D computer animation laboratory assignments. *IEEE Transactions on Learning Technologies* 7, 3 (July 2014), 280–290. https://doi.org/10.1109/TLT.2014.2340861

[139] Timotej Lazar, Martin Možina, and Ivan Bratko. 2017. Automatic extraction of AST patterns for debugging student programs. In *Artificial Intelligence in Education*, Elisabeth André, Ryan Baker, Xiangen Hu, Ma. Mercedes T. Rodrigo, and Benedict du Boulay (Eds.). Springer International, Cham, Switzerland, 162–174.

[140] Timotej Lazar, Aleksander Sadikov, and Ivan Bratko. 2018. Rewrite rules for debugging student programs in programming tutors. *IEEE Transactions on Learning Technologies* 11, 4 (Oct. 2018), 429–440. https://doi.org/10.1109/TLT.2017.2743701

[141] José Paulo Leal and Fernando Silva. 2003. Mooshak: A web-based multi-site programming contest system. *Software: Practice and Experience* 33, 6 (2003), 567–581. https://doi.org/10.1002/spe.522

[142] Teemu Lehtinen, Andre L. Santos, and Juha Sorva. 2021. Let's ask students about their programs, automatically. In *Proceedings of the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC'21)*. IEEE, Los Alamitos, CA, 467–475. https://doi.org/10.1109/ICPC52881.2021.00054

[143] Abe Leite and Saúl A. Blanco. 2020. Effects of human vs. automatic feedback on students' understanding of AI concepts and programming style. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE'20)*. ACM, New York, NY, 44–50. https://doi.org/10.1145/3328778.3366921

[144] Leo Leppänen, Juho Leinonen, and Arto Hellas. 2016. Pauses and spacing in learning to program. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling'16)*. ACM, New York, NY, 41–50. https://doi.org/10.1145/2999541.2999549

[145] Sihan Li, Xusheng Xiao, Blake Bassett, Tao Xie, and Nikolai Tillmann. 2016. Measuring code behavioral similarity for programming and software engineering education. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, New York, NY, 501–510. https://doi.org/10.1145/2889160.2889204

[146] Yue Li, Yiqing Pan, Wensheng Liu, and Xingming Zhang. 2018. An automated evaluation system for app inventor apps. In *Proceedings of the 2018 IEEE 16th International Conference on Dependable, Autonomic, and Secure Computing, the 16th International Conference on Pervasive Intelligence and Computing, and the 4th International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech'18)*. IEEE, Los Alamitos, CA, 230–235. https://doi.org/10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00048

[147] M. Lingling, Q. Xiaojie, Z. Zhihong, Z. Gang, and X. Ying. 2008. An assessment tool for assembly language programming. In *Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, Vol. 5. IEEE, Los Alamitos, CA, 882–884. https://doi.org/10.1109/CSSE.2008.111

[148] Xiao Liu, Yeoneo Kim, Junseok Cheon, and Gyun Woo. 2019. A partial grading method using pattern matching for programming assignments. In *Proceedings of the 2019 8th International Conference on Innovation, Communication, and Engineering (ICICE'19)*. IEEE, Los Alamitos, CA, 157–160. https://doi.org/10.1109/ICICE49024.2019.9117506

[149] Xiao Liu, Shuai Wang, Pei Wang, and Dinghao Wu. 2019. Automatic grading of programming assignments: An approach based on formal semantics. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET'19)*. IEEE, Los Alamitos, CA, 126–137. https://doi.org/10.1109/ICSE-SEET.2019.00022

[150] Xiao Liu and Gyun Woo. 2020. Applying code quality detection in online programming judge. In *Proceedings of the 2020 5th International Conference on Intelligent Information Technology*. ACM, New York, NY, 56–60. https://doi.org/10.1145/3385209.3385226

[151] Richard Lobb and Jenny Harlow. 2016. CodeRunner: A tool for assessing computer programming skills. *ACM Inroads* 7, 1 (Feb. 2016), 47–51. https://doi.org/10.1145/2810041

[152] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, New York, NY, 389–400. https://doi.org/10.1145/2635868.2635900

[153] Matej Madeja and Jaroslav Poruban. 2017. Automatic assessment of assignments for Android application programming courses. In *Proceedings of the 2017 IEEE 14th International Scientific Conference on Informatics*. IEEE, Los Alamitos, CA, 232–237. https://doi.org/10.1109/INFORMATICS.2017.8327252

[154] Matej Madeja and Jaroslav Poruban. 2018. Automated testing environment and assessment of assignments for Android MOOC. *Open Computer Science* 8, 1 (2018), 80–92. https://doi.org/10.1515/comp-2018-0007

[155] Christoph Matthies, Arian Treffer, and Matthias Uflacker. 2017. Prof. CI: Employing continuous integration services and GitHub workflows to teach test-driven development. In *Proceedings of the 2017 IEEE Frontiers in Education Conference (FIE'17)*. IEEE, Los Alamitos, CA, 1–8. https://doi.org/10.1109/FIE.2017.8190589

[156] Igor Mekterovic, Ljiljana Brkic, Boris Milasinovic, and Mirta Baranovic. 2020. Building a comprehensive automated programming assessment system. *IEEE Access* 8 (2020), 81154–81172. https://doi.org/10.1109/ACCESS.2020.2990980

[157] Jesús Moreno-León and Gregorio Robles. 2015. Dr. Scratch: A web tool to automatically evaluate scratch projects. In *Proceedings of the Workshop in Primary and Secondary Computing Education (WiPSCE'15)*. ACM, New York, NY, 132–133. https://doi.org/10.1145/2818314.2818338

[158] D. S. Morris. 2003. Automatic grading of student's programming assignments: An interactive process and suite of programs. In *Proceedings of the 2003 33rd Annual Frontiers in Education Conference (FIE'03)*, Vol. 3. IEEE, Los Alamitos, CA. https://doi.org/10.1109/FIE.2003.1265998

[159] Jonathan P. Munson and Joshua P. Zitovsky. 2018. Models for early identification of struggling novice programmers. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE'18)*. ACM, New York, NY, 699–704. https://doi.org/10.1145/3159450.3159476

[160] Kevin A. Naudé, Jean H. Greyling, and Dieter Vogts. 2010. Marking student programs using graph similarity. *Computers & Education* 54, 2 (2010), 545–561. https://doi.org/10.1016/j.compedu.2009.09.005

[161] Peter Naur. 1964. Automatic grading of students' ALGOL programming. *BIT* 4, 3 (Sept. 1964), 177–188. https://doi.org/10.1007/BF01956028

[162] Mark Noone and Aidan Mooney. 2018. Visual and textual programming languages: A systematic review of the literature. *Journal of Computers in Education* 5, 2 (June 2018), 149–174. https://doi.org/10.1007/s40692-018-0101-5

[163] M. Novak. 2016. Review of source-code plagiarism detection in academia. In *Proceedings of the 2016 39th International Convention on Information and Communication Technology, Electronics, and Microelectronics (MIPRO'16)*. IEEE, Los Alamitos, CA, 796–801. https://doi.org/10.1109/MIPRO.2016.7522248

[164] Atsushi Nunome, Hiroaki Hirata, Masayuki Fukuzawa, and Kiyoshi Shibayama. 2010. Development of an e-learning back-end system for code assessment in elementary programming practice. In *Proceedings of the 38th Annual SIGUCCS Fall Conference(SIGUCCS'10)*. ACM, New York, NY, 181. https://doi.org/10.1145/1878335.1878381

[165] Rainer Oechsle and Kay Barzen. 2007. Checking automatically the output of concurrent threads. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'07)*. ACM, New York, NY, 43–47. https://doi.org/10.1145/1268784.1268799

[166] University of Maryland. 2015. FindBugs—Find bugs in Java programs. *SourceForge*. Retrieved August 31, 2021 from http://findbugs.sourceforge.net/.

[167] Tony Ohmann and Imad Rahal. 2014. Efficient clustering-based source code plagiarism detection using PIY. *Knowledge and Information Systems* 43, 2 (March 2014), 445–472. https://doi.org/10.1007/s10115-014-0742-2

[168] Jonathan Osei-Owusu, Angello Astorga, Liia Butler, Tao Xie, and Geoffrey Challen. 2019. Grading-based test suite augmentation. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. IEEE, Los Alamitos, CA, 226–229. https://doi.org/10.1109/ASE.2019.00030

[169] G. Ota, Y. Morimoto, and H. Kato. 2016. Ninja code village for Scratch: Function samples/function analyser and automatic assessment of computational thinking concepts. In *Proceedings of the 2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'16)*. IEEE, Los Alamitos, CA, 238–239. https://doi.org/10.1109/VLHCC.2016.7739695

[170] José Carlos Paiva, José Paulo Leal, and Ricardo Alexandre Queirós. 2016. *Enki: A Pedagogical Services Aggregator for Learning Programming Languages*. ACM, New York, NY, 332–337. https://doi.org/10.1145/2899415.2899441

[171] Fauhat Ali Khan Panni and Abu Sayed Md. Latiful Hoque. 2020. A model for automatic partial evaluation of SQL queries. In *Proceedings of the 2020 2nd International Conference on Advanced Information and Communication Technology (ICAICT'20)*. IEEE, Los Alamitos, CA, 240–245. https://doi.org/10.1109/ICAICT51780.2020.9333475

[172] Sebastian Pape, Julian Flake, Andreas Beckmann, and Jan Jurjens. 2016. STAGE: A software tool for automatic grading of testing exercises: Case study paper. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, New York, NY, 491–500. https://doi.org/10.1145/2889160.2889203

[173] Sagar Parihar, Ziyaan Dadachanji, Praveen Kumar Singh, Rajdeep Das, Amey Karkare, and Arnab Bhattacharya. 2017. Automatic grading and feedback using program repair for introductory programming courses. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, 92–97. https://doi.org/10.1145/3059009.3059026

[174] Abhinandan H. Patil and Nandini S. Sidnal. 2014. CodeCover: Enhancement of codecover. *SIGSOFT Software Engineering Notes* 39, 1 (Feb. 2014), 1–4. https://doi.org/10.1145/2557833.2557850

[175] Isidoros Perikos, Foteini Grivokostopoulou, and Ioannis Hatzilygeroudis. 2012. Automatic marking of NL to FOL conversions. In *Computers and Advanced Technology in Education*. ACTAPRESS, Napoli, Italy. https://doi.org/10.2316/P.2012.774-070

[176] Jordi Petit, Salvador Roura, Josep Carmona, Jordi Cortadella, Jordi Duch, Omer Gimnez, Anaga Mani, et al. 2018. Jutge.org: Characteristics and experiences. *IEEE Transactions on Learning Technologies* 11, 3 (July 2018), 321–333. https://doi.org/10.1109/TLT.2017.2723389

[177] Raymond Pettit, John Homer, Kayla Holcomb, Nevan Simone, and Susan Mengel. 2015. Are automated assessment tools helpful in programming courses? In *Proceedings of the 2015 ASEE Annual Conference and Exposition Proceedings*. 1–20. https://doi.org/10.18260/p.23569

[178] Matthew Peveler, Evan Maicus, and Barbara Cutler. 2019. Comparing jailed sandboxes vs containers within an autograding system. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE'19)*. ACM, New York, NY, 139–145. https://doi.org/10.1145/3287324.3287507

[179] Matthew Peveler, Evan Maicus, and Barbara Cutler. 2020. Automated and manual grading of web-based assignments. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, 1373–1373. https://doi.org/10.1145/3328778.3372682

[180] Matthew Peveler, Jeramey Tyler, Samuel Breese, Barbara Cutler, and Ana Milanova. 2017. Submitty: An open source, highly-configurable platform for grading of programming assignments (abstract only). In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE'17)*. ACM, New York, NY, 641. https://doi.org/10.1145/3017680.3022384

[181] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the 2015 2nd ACM Conference on Learning @ Scale (L@S'15)*. ACM, New York, NY, 195–204. https://doi.org/10.1145/2724660.2724668

[182] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. 2012. Modeling how students learn to program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education (SIGCSE'12)*. ACM, New York, NY, 153–160. https://doi.org/10.1145/2157136.2157182

[183] Vreda Pieterse. 2013. Automated assessment of programming assignments. In *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research (CSERC'13)*. 45–56. https://doi.org/10.5555/2541917.2541921

[184] Emilia Pietrikova, Jan Juhar, and Jana Stastna. 2015. Towards automated assessment in game-creative programming courses. In *Proceedings of the 2015 13th International Conference on Emerging eLearning Technologies and Applications (ICETA'15)*. IEEE, Los Alamitos, CA, 1–6. https://doi.org/10.1109/ICETA.2015.7558505

[185] PMD. 2020. PMD Open Source Project. Retrieved August 31, 2021 from https://pmd.github.io/.

[186] Jonathan Y. H. Poon, Kazunari Sugiyama, Yee Fan Tan, and Min-Yen Kan. 2012. Instructor-centric source code plagiarism detection and plagiarism corpus. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'12)*. ACM, New York, NY, 122–127. https://doi.org/10.1145/2325296.2325328

[187] Bruno O. Prado, Kalil A. Bispo, and Raul Andrade. 2018. X9: An obfuscation resilient approach for source code plagiarism detection in virtual learning environments. In *Proceedings of the 20th International Conference on Enterprise Information Systems*, Slimane Hammoudi, Michal Smialek, Olivier Camp, and Joaquim Filipe (Eds.), Vol. 1. INSTICC, SciTePress, Funchal, Madeira, Portugal, 517–524. https://doi.org/10.5220/0006668705170524

[188] James Prather, Raymond Pettit, Brett A. Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. 2019. First things first: Providing metacognitive scaffolding for interpreting problem prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE'19)*. ACM, New York, NY, 531–537. https://doi.org/10.1145/3287324.3287374

[189] James Prather, Raymond Pettit, Kayla McMurry, Alani Peters, John Homer, and Maxine Cohen. 2018. Metacognitive difficulties faced by novice programmers in automated assessment tools. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER'18)*. ACM, New York, NY, 41–50. https://doi.org/10.1145/3230977.3230981

[190] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2000. *JPlag: Finding Plagiarisms among a Set of Programs*. Technical Report 1. Department of Informatics, University of Karlsruhe. https://doi.org/10.5445/IR/542000

[191] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2002. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science* 8, 11 (2002), 1016–1038. https://doi.org/10.3217/jucs-008-11-1016

[192] Thomas W. Price, Yihuan Dong, and Tiffany Barnes. 2016. Generating data-driven hints for open-ended programming. *International Educational Data Mining Society* 9 (2016), 191–198.

[193] Thomas W. Price, David Hovemeyer, Kelly Rivers, Ge Gao, Austin Cory Bart, Ayaan M. Kazerouni, Brett A. Becker, et al. 2020. ProgSnap2: A flexible format for programming process data. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'20)*. ACM, New York, NY, 356–362. https://doi.org/10.1145/3341525.3387373

[194] Martin Partel, Matti Luukkainen, Arto Vihavainen, and Thomas Vikberg. 2013. Test My Code. *International Journal of Technology Enhanced Learning* 5, 3–4 (2013), 271. https://doi.org/10.1504/IJTEL.2013.059495

[195] Ricardo Queirós, José Paulo Leal, and José Carlos Paiva. 2016. Integrating rich learning applications in LMS. In *State-of-the-Art and Future Directions of Smart Learning*, Yanyan Li, Maiga Chang, Milos Kravcik, Elvira Popescu, Ronghuai Huang, Kinshuk, and Nian-Shing Chen (Eds.). Springer Singapore, Singapore, 381–386.

[196] Faqih Rabbani and Oscar Karnalim. 2017. Detecting source code plagiarism on .NET programming languages using low-level representation and adaptive local alignment. *Journal of Information and Organizational Sciences* 41, 1 (June 2017), 1–19. https://doi.org/10.31341/jios.41.1.7

[197] Khirulnizam Rahman and Md. Jan Nordin. 2007. A review on the static analysis approach in the automated programming assessment systems. In *Proceedings of the 2007 National Conference on Programming*. 1–15.

[198] Kenneth A. Reek. 1989. The TRY system—Or—How to avoid testing student programs. In *Proceedings of the 20th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'89)*. ACM, New York, NY, 112–116. https://doi.org/10.1145/65293.71198

[199] Michael J. Rees. 1982. Automatic assessment aids for pascal programs. *ACM SIGPLAN Notices* 17, 10 (Oct. 1982), 33–42. https://doi.org/10.1145/948086.948088

[200] Tobias Reischmann and Breno Menezes. 2019. Application of swarm-intelligent methods to optimize error-tolerant graph matching for automatic e-assessment. In *Proceedings of the 2019 IEEE Latin American Conference on Computational Intelligence (LA-CCI'19)*. IEEE, Los Alamitos, CA, 1–6. https://doi.org/10.1109/LA-CCI47412.2019.9037041

[201] Sasko Ristov, Marjan Gusev, Goce Armenski, Krste Bozinoski, and Goran Velkoski. 2013. Architecture and organization of e-assessment cloud solution. In *Proceedings of the 2013 IEEE Global Engineering Education Conference (EDUCON'13)*. IEEE, Los Alamitos, CA, 736–743. https://doi.org/10.1109/educon.2013.6530189

[202] Kelly Rivers and Kenneth R. Koedinger. 2017. Data-driven hint generation in vast solution spaces: A self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 37–64. https://doi.org/10.1007/s40593-015-0070-z

[203] Philip E. Robinson and Johnson Carroll. 2017. An online learning platform for teaching, learning, and assessment of programming. In *Proceedings of the 2017 IEEE Global Engineering Education Conference (EDUCON'17)*. IEEE, Los Alamitos, CA, 547–556. https://doi.org/10.1109/EDUCON.2017.7942900

[204] R. Romli, S. Sulaiman, and K. Z. Zamli. 2010. Automatic programming assessment and test data generation: A review on its approaches. In *Proceedings of the 2010 International Symposium on Information Technology*, Vol. 3. IEEE, Los Alamitos, CA, 1186–1192. https://doi.org/10.1109/ITSIM.2010.5561488

[205] Vlad Roubtsov. 2006. EMMA: A free Java code coverage tool. *SourceForge*. Retrieved August 31, 2021 from http://emma.sourceforge.net/index.html.

[206] Manuel Rubio-Sanchez, Paivi Kinnunen, Cristobal Pareja-Flores, and J. Angel Velazquez-Iturbide. 2014. Student perception and usage of an automated programming assessment tool. *Computers in Human Behavior* 31 (Feb. 2014), 453–460. https://doi.org/10.1016/j.chb.2013.04.001

[207] SonarSource S. A. 2020. SonarQube. *SonarSource S.A.* Retrieved August 31, 2021 from https://www.sonarqube.org/.

[208] A. Sanna, F. Lamberti, G. Paravati, and C. Demartini. 2012. Automatic assessment of 3D modeling exams. *IEEE Transactions on Learning Technologies* 5, 1 (2012), 2–10. https://doi.org/10.1109/TLT.2011.4

[209] V. S. Shekhar, A. Agarwalla, A. Agarwal, B. Nitish, and V. Kumar. 2014. Enhancing JFLAP with automata construction problems and automated feedback. In *Proceedings of the 2014 7th International Conference on Contemporary Computing (IC3'14)*. IEEE, Los Alamitos, CA, 19–23. https://doi.org/10.1109/IC3.2014.6897141

[210] Clóvis Daniel Souza Silva, Leonardo Ferreira da Costa, Leonardo Sampaio Rocha, and Gerardo Valdísio Rodrigues Viana. 2020. KNN applied to PDG for source code similarity classification. In *Intelligent Systems*, Ricardo Cerri and Ronaldo C. Prati (Eds.). Springer, Cham, Switzerland, 471–482.

[211] H. Simanjuntak. 2015. Proposed framework for automatic grading system of ER diagram. In *Proceedings of the 2015 7th International Conference on Information Technology and Electrical Engineering (ICITEE'15)*. IEEE, Los Alamitos, CA, 141–146. https://doi.org/10.1109/ICITEED.2015.7408930

[212] Ryan W. Sims. 2012. *Secure Execution of Student Code*. Technical Report. Department of Computer Science, University of Maryland.

[213] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, New York, NY, 15. https://doi.org/10.1145/2491956.2462195

[214] Antonio Carvalho Siochi and William Randall Hardy. 2015. WebWolf: Towards a simple framework for automated assessment of webpage assignments in an introductory web programming class. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, 84–89. https://doi.org/10.1145/2676723.2677217

[215] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15, 5 (Oct. 2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

[216] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*. ACM, New York, NY, 404–415. https://doi.org/10.1145/1168857.1168907

[217] Igor Solecki, João Porto, Nathalia da Cruz Alves, Christiane Gresse von Wangenheim, Jean Hauck, and Adriano Ferreti Borgatto. 2020. Automated assessment of the visual design of Android apps developed with App Inventor. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE'20)*. ACM, New York, NY, 51–57. https://doi.org/10.1145/3328778.3366868

[218] Igor da Silva Solecki, João Vitor Araujo Porto, Karla Aparecida Justen, Nathalia da Cruz Alves, Christiane Gresse von Wangenheim, Adriano Ferreti Borgatto, and Jean Carlo Rossa Hauck. 2019. CodeMaster UI Design—App Inventor: A rubric for the assessment of the interface design of Android apps developed with App Inventor. In *Proceedings of the 18th Brazilian Symposium on Human Factors in Computing Systems (IHC'19)*. ACM, New York, NY, Article 17, 10 pages. https://doi.org/10.1145/3357155.3358463

[219] Dowon Song, Myungho Lee, and Hakjoo Oh. 2019. Automatic and scalable detection of logical errors in functional programming assignments. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, (Oct. 2019), Article 188, 30 pages. https://doi.org/10.1145/3360614

[220] Draylson M. Souza, Katia R. Felizardo, and Ellen F. Barbosa. 2016. A systematic literature review of assessment tools for programming assignments. In *Proceedings of the 2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET'16)*. IEEE, Los Alamitos, CA, 147–156. https://doi.org/10.1109/CSEET.2016.48

[221] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K. Hollingsworth, and Nelson Padua-Perez. 2006. Experiences with Marmoset: Designing and using an advanced submission and testing system for programming courses. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE'06)*. ACM, New York, NY, 13–17. https://doi.org/10.1145/1140124.1140131

[222] David Spencer. 2014. Top 10 common database security issues. *NCC Group*. Retrieved December 23, 2020 from https://www.nccgroup.com/uk/about-us/newsroom-and-events/blogs/2014/july/top-10-common-database-security-issues/.

[223] SpotBugs. 2020. SpotBugs. Retrieved August 31, 2021 from http://spotbugs.github.io/.

[224] Computerworld UK Staff. 2020. Top software failures in recent history. *Computerworld*. Retrieved December 22, 2020 from https://www.computerworld.com/article/3412197/top-software-failures-in-recent-history.html.

[225] T. Staubitz, H. Klement, J. Renz, R. Teusner, and C. Meinel. 2015. Towards practical programming exercises and automated assessment in massive open online courses. In *Proceedings of the 2015 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE'15)*. IEEE, Los Alamitos, CA, 23–30. https://doi.org/10.1109/TALE.2015.7386010

[226] Thomas Staubitz, Hauke Klement, Ralf Teusner, Jan Renz, and Christoph Meinel. 2016. CodeOcean—A versatile platform for practical programming excercises in online environments. In *Proceedings of the 2016 IEEE Global Engineering Education Conference (EDUCON'16)*. IEEE, Los Alamitos, CA, 314–323. https://doi.org/10.1109/EDUCON.2016.7474573

[227] Michael Striewe. 2016. An architecture for modular grading and feedback generation for complex exercises. *Science of Computer Programming* 129 (Nov. 2016), 35–47. https://doi.org/10.1016/j.scico.2016.02.009

[228] Michael Striewe and Michael Goedicke. 2014. Automated assessment of UML activity diagrams. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education (ITiCSE'14)*. ACM, Los Alamitos, CA, 336. https://doi.org/10.1145/2591708.2602657

[229] Lisan Sulistiani and Oscar Karnalim. 2019. ES-Plag: Efficient and sensitive source code plagiarism detection tool for academic environment. *Computer Applications in Engineering Education* 27, 1 (2019), 166–182. https://doi.org/10.1002/cae.22066

[230] Kelvin Sung and Arjmand Samuel. 2014. Mobile application development classes for the mobile era. In *Proceedings of the 2014 Conference on Innovation and Technology in Computer Science Education (ITiCSE'14)*. ACM, New York, NY, 141–146. https://doi.org/10.1145/2591708.2591710

[231] Mate' Sztipanovits, Kai Qian, and Xiang Fu. 2008. The automated web application testing (AWAT) system. In *Proceedings of the 46th Annual Southeast Regional Conference (ACM-SE'08)*. ACM, New York, NY, 88–93. https://doi.org/10.1145/1593105.1593128

[232] Mike Talbot, Katharina Geldreich, Julia Sommer, and Peter Hubwieser. 2020. Re-use of programming patterns or problem solving? Representation of Scratch programs by TGraphs to support static code analysis. In *Proceedings of the 15th Workshop on Primary and Secondary Computing Education (WiPSCE'20)*. ACM, New York, NY, Article 16, 10 pages. https://doi.org/10.1145/3421590.3421604

[233] Matti Tedre. 2020. From a black art to a school subject: Computing education's search for status. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'20)*. ACM, New York, NY, 3–4. https://doi.org/10.1145/3341525.3394983

[234] J. F. Temperly and Barry W. Smith. 1968. A grading procedure for PL/1 student exercises. *Computer Journal* 10, 4 (Feb. 1968), 368–373. https://doi.org/10.1093/comjnl/10.4.368

[235] Matthew Thornton, Stephen H. Edwards, Roy P. Tan, and Manuel A. Pérez-Quiñones. 2008. Supporting student-written tests of GUI programs. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'08)*. ACM, New York, NY, 537–541. https://doi.org/10.1145/1352135.1352316

[236] Nghi Truong, Paul Roe, and Peter Bancroft. 2004. Static analysis of students' Java programs. In *Proceedings of the 6th Australasian Conference on Computing Education—Volume 30 (ACE'04)*. 317–325. https://doi.org/10.5555/979968.980011

[237] Farhan Ullah, Junfeng Wang, Muhammad Farhan, Sohail Jabbar, Zhiming Wu, and Shehzad Khalid. 2018. Plagiarism detection in students' programming assignments based on semantics: Multimedia e-learning based smart assessment methodology. *Multimedia Tools and Applications* 79, 13–14 (March 2018), 8581–8598. https://doi.org/10.1007/s11042-018-5827-6

[238] Leo C. Ureel II and Charles Wallace. 2019. Automated critique of early programming antipatterns. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, 738–744. https://doi.org/10.1145/3287324.3287463

[239] Vinay Vachharajani and Jyoti Pareek. 2014. A proposed architecture for automated assessment of use case diagrams. *International Journal of Computer Applications* 108, 4 (Dec. 2014), 35–40. https://doi.org/10.5120/18902-0193

[240] Patricia Bylebyl Van Verth. 1985. *A System for Automatically Grading Program Quality (Metrics, Software Metrics, Program Complexity)*. Ph.D. Dissertation. State University of NewYork at Buffalo.

[241] Brad Vander Zanden and Michael W. Berry. 2013. Improving automatic code assessment. *Journal of Computing Sciences in Colleges* 29, 2 (Dec. 2013), 162–168. https://doi.org/10.5555/2535418.2535443

[242] B. Vesin, A. Klasnja-Millicevic, and M. Ivanovic. 2013. Improving testing abilities of a programming tutoring system. In *Proceedings of the 2013 17th International Conference on System Theory, Control, and Computing (ICSTCC'13)*. IEEE, Los Alamitos, CA, 669–673. https://doi.org/10.1109/ICSTCC.2013.6689037

[243] Arto Vihavainen, Matti Luukkainen, and Petri Ihantola. 2014. Analysis of source code snapshot granularity levels. In *Proceedings of the 15th Annual Conference on Information Technology Education (SIGITE'14)*. ACM, New York, NY, 21–26. https://doi.org/10.1145/2656450.2656473

[244] Nickolay Viuginov, Petr Grachev, and Andrey Filchenkov. 2020. A machine learning based plagiarism detection in source code. In *Proceedings of the 2020 3rd International Conference on Algorithms, Computing, and Artificial Intelligence (ACAI'20)*. ACM, New York, NY, Article 93, 6 pages. https://doi.org/10.1145/3446132.3446420

[245] Urs Von Matt. 1994. Kassandra: The automatic grading system. *ACM SIGCUE Outlook* 22, 1 (1994), 26–40. https://doi.org/10.1145/182107.182101

[246] Christiane Gresse Von Wangenheim, Jean C. R. Hauck, Matheus Faustino Demetrio, Rafael Pelle, Nathalia da Cruz Alves, Heliziane Barbosa, and Luiz Felipe Azevedo. 2018. CodeMaster—Automatic assessment and grading of App Inventor and Snap! programs. *Informatics in Education* 17, 1 (2018), 117–150. https://doi.org/10.15388/infedu.2018.08

[247] Milena Vujosevic-Janicic, Mladen Nikolic, Dusan Tosic, and Viktor Kuncak. 2013. Software verification and graph similarity for automated evaluation of students' assignments. *Information and Software Technology* 55, 6 (June 2013), 1004–1016. https://doi.org/10.1016/j.infsof.2012.12.005

[248] Tiantian Wang, Xiaohong Su, Yuying Wang, and Peijun Ma. 2007. Semantic similarity-based grading of student programs. *Information and Software Technology* 49, 2 (Feb. 2007), 99–107. https://doi.org/10.1016/j.infsof.2006.03.001

[249] Weichao Wang, Zhaopeng Meng, Zan Wang, Shuang Liu, and Jianye Hao. 2019. LoopFix: An approach to automatic repair of buggy loops. *Journal of Systems and Software* 156 (Oct. 2019), 100–112. https://doi.org/10.1016/j.jss.2019.06.076

[250] Kevin Waugh, Pete Thomas, and Neil Smith. 2004. Toward the automated assessment of entity-relationship diagrams. In *Proceedings of the 2nd Workshop of the Learning and Teaching Support Network—Information and Computer Science (TLAD'04)*. 1–6. http://oro.open.ac.uk/2455/

[251] Burkhard C. Wünsche, Zhen Chen, Lindsay Shaw, Thomas Suselo, Kai-Cheung Leung, Davis Dimalen, Wannes van der Mark, Andrew Luxton-Reilly, and Richard Lobb. 2018. Automatic assessment of OpenGL computer graphics assignments. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ACM, New York, NY, 81–86. https://doi.org/10.1145/3197091.3197112

[252] Yi-Xiang Yan, Jung-Pin Wu, Bao-An Nguyen, and Hsi-Min Chen. 2020. The impact of iterative assessment system on programming learning behavior. In *Proceedings of the 2020 9th International Conference on Educational and Information Technology*. ACM, New York, NY, 89–94. https://doi.org/10.1145/3383923.3383939

[253] S. Yassine, S. Kadry, and M. Sicilia. 2016. A framework for learning analytics in Moodle for assessing course outcomes. In *Proceedings of the 2016 IEEE Global Engineering Education Conference (EDUCON'16)*. IEEE, Los Alamitos, CA, 261–266. https://doi.org/10.1109/EDUCON.2016.7474563

[254] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, New York, NY, 740–751. https://doi.org/10.1145/3106237.3106262

[255] Y. T. Yu, C. M. Tang, and C. K. Poon. 2017. Enhancing an automated system for assessment of student programs using the token pattern approach. In *Proceedings of the 2017 IEEE 6th International Conference on Teaching, Assessment, and Learning for Engineering (TALE'17)*. IEEE, Los Alamitos, CA, 406–413. https://doi.org/10.1109/TALE.2017.8252370

[256] Fangfang Zhang, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Program logic based software plagiarism detection. In *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, Los Alamitos, CA, 66–77. https://doi.org/10.1109/ISSRE.2014.18

[257] Jingling Zhao, Kunfeng Xia, Yilun Fu, and Baojiang Cui. 2015. An AST-based code plagiarism detection algorithm. In *Proceedings of the 2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA'15)*. IEEE, Los Alamitos, CA, 178–182. https://doi.org/10.1109/BWCCA.2015.52

[258] Soundous Zougari, Mariam Tanana, and Abdelouahid Lyhyaoui. 2016. Hybrid assessment method for programming assignments. In *Proceedings of the 2016 4th IEEE International Colloquium on Information Science and Technology (CiSt'16)*. IEEE, Los Alamitos, CA, 564–569. https://doi.org/10.1109/CIST.2016.7805112

[259] Zoran Đurić and Dragan Gašević. 2012. A source code similarity system for plagiarism detection. *Computer Journal* 56, 1 (March 2012), 70–86. https://doi.org/10.1093/comjnl/bxs018

Received February 2021; revised October 2021; accepted January 2022