*Review*

# Automated Code Assessment for Education: Review, Classification and Perspectives on Techniques and Tools

**Sébastien Combéfis** [1,2]

[1] Institut Technique Supérieur Cardinal Mercier (ITSCM), 1030 Schaerbeek, Belgium; sebastien@combefis.be
[2] École Pratique des Hautes Études Commerciales (EPHEC), 1200 Woluwe-Saint-Lambert, Belgium

**Abstract:** Automatically assessing code for learning purposes is a challenging goal to achieve. On-site courses and online ones developed for distance learning both require automated ways to grade learners' programs to be able to scale and manage a large public with a limited teaching staff. This paper reviews recent automated code assessment systems. It proposes a systematic review of the possible analyses they can perform with the associated techniques, the kinds of produced feedback and the ways they are integrated in the learning process. It then discusses the key challenges for the development of new automated code assessment systems and the interaction with human grading. In conclusion, the paper draws several recommendations for new research directions and for possible improvements for automatic code assessment.

**Keywords:** automated code assessment; code graders; code analysis techniques

**Citation:** Combéfis, S. Automated Code Assessment for Education: Review, Classification and Perspectives. *Software* **2022**, *1*, 3–30. https://doi.org/10.3390/software1010002

Academic Editors: Paolino Di Felice and Tommi Mikkonen

Received: 6 December 2021
Accepted: 25 January 2022
Published: 8 February 2022

## 1. Introduction

Nowadays, computer-science-related courses are delivered in many kinds of training. In particular, programming courses are being taught to very large audiences, starting from pupils in primary and secondary schools to young adults in higher education, including people following lifelong learning programs. Instructors facing these rapidly growing audiences and the associated massive amount of code produced by learners to grade are struggling with human resource issues. This resulted in the development of semi or fully automated tools to assist them for code assessment.

### *1.1. Motivations*

The principal reason that triggered the development of automated code assessment systems is that the number of people who are learning programming increases while the size of teaching staff stays constant and small. Concrete development being necessary to achieve computer programming skills, this for sure led to significant challenges for instructors who need to evaluate codes produced by their learners.

Having support to automatically judge codes is important to save human resources and to relieve instructors from some lengthy tasks such as grading. Marking learners' programs constitutes a substantial source of workloads in computer science courses, and providing them timely and relevant feedback is crucial. E-learning has also been taking additional roles in the current models of education, in particular since the arrival of MOOCs and more recently with the COVID-19 pandemic. These last two observations both increase the need for powerful and efficient systems to automatically evaluate codes. They should be the best ways to extract syntactic and semantic features of the correct program to be able to produce relevant assessments that support learning. Judging for wrong programs is also critical for these systems, allowing them to generate the best possible feedback to help learners understand their faults and make progress.

Both the large number of learners and the fact that computer science and programming has spread widely into society contribute to the crucial need for semi or fully automated systems to assess codes in the most human way possible to support learning.

## *1.2. Research Questions*

This paper presents a review of techniques and tools that can be used to automatically assess codes for educational purposes. The presented review has been realised to answer three main research questions:

- What kinds of automatic code assessments are feasible?
- To what extent can an automated code assessment tool generate results that are as good as those obtained manually by a human grading process?
- What are the key challenges related to automatic code assessment?

The first question instructors willing to use automated code assessment techniques may ask themselves is whether the grading they are currently manually performing can be automatically carried out. Then, depending on whether they need summative or formative evaluations, instructors may want to know what is the expected quality of the results produced by these tools. Finally, researchers are interested in the key challenges related to the development of novel techniques and tools. The three research questions aim at providing answer elements to these highlighted concerns.

The proposed review also aims to encourage researchers to better collaborate. They have indeed been designing tools for many years, often in isolation, while the features needed by instructors are, for most of the developed systems, similar. Compared to existing reviews, this paper proposes a systematic analysis which results in recommendations and research perspectives towards more unified development and easier sharing possibilities between systems.

## *1.3. Methodology*

The proposed approach used to bring answer elements to the three raised research questions is based on an extensive literature review from papers found following several strategies. The primary sources are Google Scholar and various widespread publishers including ACM, IEEE, Springer and MDPI. The main keywords used in their online search engines are "automated code assessment", "automatic code grader", "automatic programming assessment" and "automated assessment computer education." Relevant references of the found papers have also been examined and included in the collection.

From this, only those written in English and published from the year 2000 have been kept for the work presented in this paper, no matter their type and whether they have been peer-reviewed or not. In addition, the focus being put on code assessment techniques and tools that can or may be used for educational purposes, only relevant references have been kept. For example, papers focused on algorithms, techniques or tools to analyse code but not directly related to automatic code assessment for educational purposes have been put aside. Selected papers have then been categorised based on their abstracts into three categories: review or survey, tool presentation and others.

## *1.4. Related Work*

To obtain a holistic overview of research works related to automated code assessment in education, review and survey papers have first been analysed. Eighteen references, published between 2005 and 2021 and covering several aspects of techniques and tools to automatically assess codes, have been identified [1–18]. The review of these papers made it possible to highlight six main concerns related to the development of automated code assessment systems:

1. the kinds of coding/program aspects that should be assessed;
2. the methods and techniques used to analyse the code;
3. the types of feedback that are presented to learners and instructors;
4. the kinds of systems developed to support the automated code assessment;
5. the ways they are integrated in the learning process and used in education;
6. the quality and impact of the automatically produced assessments.

For computer science and software engineering related courses, problems and assignments are usually essential educational elements. Assessing these assignments is crucial in the learning process. All the examined reviews agree that assessment related activities place a high demand on instructors' time and resources. The development of automated ways to assist instructors for these assessments started in 1960, with the main goal of reducing the burden on them while still helping learners make progress [19].

Automatically assessing codes means being able to identify defects that are non-desirable and decrease their quality. The examined reviews and survey papers show that three kinds of error are of interest, namely, syntax, runtime and logic errors [2,5,7]. They have to be targeted by the assessments as they are aligned with the kinds of defects that good programmers should avoid. In addition to code correctness, several reviews highlight the difference between functional and non-functional aspects [12]. Some systems do check style aspects [2,7,14], code quality with metrics [2,3,7,8], efficiency [2,8] and GUI testing [14]. Another difficulty is plagiarism, which can be tackled with similarity analyses [7,11]. The majority of the reviews are focused on programming and not on other computer-science-related skills. A few of them nonetheless mention techniques to assess learners' testing skills [2,8,14].

Regarding the methods and techniques used for automated code assessment, two kinds of analyses are predominant in the examined review and survey papers, namely, static and dynamic ones [2,3,6–8,15,18]. More advanced methods are based on those used in software engineering, in particular, in the software testing [5,9] and software quality fields [5]. When working with test-based assessments, having good test data is important [11], in particular if they are automatically generated [10]. The level on which the analyses are performed is also important to determine. For example, they can be achieved on the source code or bytecode, and on a tree or graph representation of the code [13]. Finally, another set of techniques used relies on the presence of a reference correct answer for the assignments to be worked on by learners [5].

A key feature of automatic assessment tools when used for education is the extraction of a measure comparing submitted programs to a solution with respect to the teaching goals [2]. To explain this measure to both learners and instructors, feedback resulting from the assessments is critical. Having systems capable of producing good feedback, especially formative ones, is a key element for being able to use them in educational settings [11]. More specifically, good feedback should explain to learners how to fix the remaining problems and take a next step for their assignments [16]. Developed systems should be opportunities to improve the level of feedback provided to learners [12]. Being able to automate feedback generation helps to make the learning process better, as feedback acts as a constant motivator [18]. Finally, in some settings, instant feedback is needed for learners to immediately know the mistakes they made to improve themselves and not repeat the errors again [7,8].

Several reviews are focused on the tools developed to support automatic assessment of code [6,12,18]. They mainly highlight the big number of diverse possible features to have on one side and the difficulty to develop them on the other side. The most frequently mentioned features include support for assessment, evaluation, grading and for the management of programming exercises [2,7]. Three main generations of tools are discussed in [1]: early systems focused on simple analyses with short reports based on graders, tool-oriented ones built on test engines and other tools such as style checkers and, finally, web-oriented tools. Developing such platforms requires taking into account numerous concrete aspects such as architecture choices, programming languages, technologies used, etc. [12]. The kind of tool is also important, generally being a library, a standalone software or a plugin to easily integrate within learning management systems [9,13,17]. Finally, the security of these platforms must be a key concern, and programs should be executed in safe sandboxes, as malicious code can be submitted by learners [9,11].

All the reviews raise the dramatic increase in the number of students enrolled in traditional and distance learning environments and who are being taught programming, resulting in the need of automatic assessments tools to assist instructors [3,12,18]. Regarding the specific use in education, such systems can be seen as learning engines improving learners' motivation, progression, self-assessment abilities and computer-science-related skills. The purpose of these platforms is to try to measure whether instructors' learning goals have been met [9]. They can either be centred on instructors who have to produce grades or on learners needing to practice and improve their skills [17,18]. Automated code assessment tools can both be used for formative or summative assessments in a fully or semi-automatic setting or even a manual one [5,9,17,18]. In the case of formative assessment, such platforms are an opportunity to give intensive sets of exercises for learners to train [7] and to move to continuous evaluation for a course [9]. In both cases, the resubmission policy is a subject of discussion, with propositions for unlimited submissions for formative assessments [9,11]. They can be used with different kinds of pedagogical approaches, such as with distance education or MOOCs [11,14] or based on competencies [3]. Finally, as this is the case without automated code assessment platforms, having high-quality assignments remains crucial [14].

Finally, very little research and analysis have been specifically conducted on the quality of the produced assessments. However, the examined reviews all agree on the fact that the assessment and the associated feedback must support the learning process.

The six main categories identified from the examined review and survey papers structure the remainder of this paper. Sections 2–4 are about the six concerns and detail results described in more specific research papers covering them. Then, Section 5 discusses how the review presented in this paper provides answer elements to the three raised research questions. It also provides some advice to guide future research and development in this field. Finally, the last section concludes the paper and identifies perspectives and possible future work.

## 2. Automated Code Assessment

Assessing a code for education generally means to review it to be able to either assign a mark to it which reflects evaluation criteria or to produce feedback to learners to support their learning. These aspects are not incompatible, as good assessments typically combine both of them, with feedback explaining the mark. A (fully) automated code assessment is one being completely computer-controlled, while a semi-automated assessment is only partially computer-assisted and requires human intervention.

### *2.1. Code and Program Aspects*

Automated code assessment techniques can be used to assess several aspects of a code or a program. Five levels that are mentioned in the literature survey conducted for the work presented in this paper can be considered.

A first aspect to examine is whether programs compile or not. To be more general, the goal is to ensure that a code is *syntactically correct*. A second level of verification that is sometimes conducted is related to *plagiarism*. The goal is to be sure that the submitted codes are authentic and have not been copied. A third aspect, a large part of the existing research being focused on it, is *code functional correctness*. The objective is to verify that the results produced by the execution of assessed codes agree with the expected solution described by any kind of specifications. A fourth level consists of assessing *code performance* aspects, such as the execution time or the memory consumption. Finally, the last level is related to *code quality* aspects, including style considerations.

The different checked aspects are usually examined following the order of the five levels, since each of them relies on the fact that the one below succeeded, as summarised by Figure 1. The order in which the two first layers are considered may be switched if anti-plagiarism is to be used first to quickly eliminate codes that must not be assessed. The second and fourth layers are often not present in simpler assessment systems.

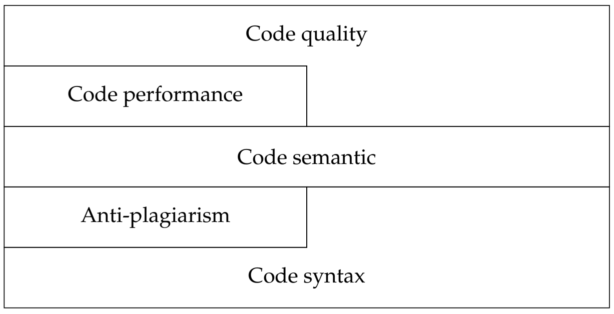

**Figure 1.** Five levels of aspects to check can generally be made on a code submission by automated code assessments tools, following the sequence presented in this figure, starting from the bottom.

#### 2.1.1. Code Syntax

Just checking that a code submission is syntactically correct is a straightforward task, as it can be performed by compilers. Instructors may want to be able to enforce additional constraints to the syntactic rules of the programming languages. In [20], the authors propose to use regular expressions to analyse the correct use of syntactic constructs, such as using semicolons and not commas in Java for loops. In [21], the authors propose another kind of syntactic check, referred to as a *keyword search*, which verifies the existence or absence of specified words. This latter approach can be generalised to ensure that only authorised statements or constructs are used. Such additional checks make it possible to have two levels of syntactic correctness. A code can be correct according to the programming language specifications and also in accordance with additional constraints. Using both kinds of checks may be relevant depending on the context.

#### 2.1.2. Anti-Plagiarism

An anti-plagiarism feature is used to ensure that codes submitted by learners are not the same or similar to other submissions [22–24]. Different kinds of sources can be examined to identify whether submissions have not been copied. Submitted codes may have been copied locally from other learners in the same learning group or from another one in the current edition of a course or from a previous year. In [25], the authors propose to compare codes with the history of submissions and to send incident reports to instructors in case of a match. Codes publicly available on the Internet can also been considered. In [26], the authors propose to perform a search on Google for lines or paragraphs and then compute a similarity index between the found codes and the submitted one. More recent techniques are not directly focused on the code but rather try to identify authorship to detect probable plagiarism [27] or to identify similar code style [28].

The plagiarism detection feature is important given the existing research evidence that academic dishonesty does exist, which is also true among computing students [22]. However, it is not easy to precisely define what *source code plagiarism* is [29]. Detecting it may be difficult, in particular for assignments for which answers are simple and short codes. They are, indeed, often similar, since there are not always many different ways to answer a simple assignment. Finally, a last difficulty arises when learners are taking code made available under a licence that allows them to use it. In this case, instructors must check the code has been used in accordance with the licence conditions [30].

Another feature related to plagiarism is code *deobfuscation*. Fraudulent learners that want to try bypassing anti-plagiarism checks are indeed usually obfuscating their codes to evade detection [31].

#### 2.1.3. Code Semantic

Checks related to the semantic of code submissions usually require executing, simulating or modelling their execution. The goal is to measure how well the submitted code does what it is meant to do. In [32], the authors propose a system to automatically assess that some functionalities have been correctly implemented as an Android mobile application. In [33], submitted codes are run against tests to check whether they conform to the requirements defined by instructors. Meeting the requirements generally means successfully passing a set of *tests* checking for functional validity. The expected result of a code also includes specific behaviours such as entering an infinite loop, throwing an exception or producing a specified side effect.

Other approaches aim at assessing programs that produce, as a result, an interactive and visual output from code, such as a *graphical user interface* [34,35] or data *visualisation* such as charts [36]. The visual rendering of the result has to be checked according to the expected one. If any interactions should be possible on the produced result, they also have to be checked. Finally, the execution of submitted codes is sometimes not necessary to check for code semantic aspects. In [37], the code of programs written in assembly language is analysed to check whether the produced output are stored in a memory area with the correct type required by instructors.

### 2.1.4. Code Performance

When writing code, another aspect that is worth evaluating is the performances, should it be the time needed for the run or the memory used during its execution. Measuring *time and memory* consumption is very usual for automatic code graders used for programming competitions but less present for educational usage. In [38,39], the authors are monitoring the execution time and memory consumption to check that constraints specified by instructors are satisfied. In [40], the authors measure the loading and running times of JavaScript programs to take these measures into account for the assessment.

### 2.1.5. Code Quality

Code quality includes many kinds of aspects that can be checked, such as *code style*, readability, maintainability, complexity, etc. Despite serious problems that may arise in software systems due to low quality programs [41] and its importance in industry, code quality is generally overlooked in first-years programming courses [42]. The focus is indeed too often only on functional correctness. Having automated techniques to assess code quality related aspects may help to make students aware of them early on if the produced feedback explains the results of the assessments [43].

In [42], the authors propose an automated system to check whether Java programs violate any *coding conventions* from a given set. This can be used to ensure that the readability of learners' codes is as high as possible, increasing their maintainability. In [44], the authors are moving beyond mechanical coding convention checks and propose to assess the effective use of programming idioms. In [40], the authors are assessing several code quality related aspects of JavaScript programs such as the style, programming errors and complexity with *metrics* using industrial tools. In [45], the authors also rely on industrial tools to analyse the style of C, Java and PHP programs. In [46], C++ programs are assessed with respect to a set of coding rules to encourage learners to produce high quality codes. For this purpose, a common C++ programming style guide has been created for instructors and learners. In [47], the robustness and security of web-based applications are assessed with tests on input validation and on data persistence against crashes. In [48], the authors propose to assess the robustness of a web page, that is, whether its layout resists to the addition of new content.

### 2.1.6. Other Aspects

Automatic assessment is not always directly related to the code but can be conducted on other artefacts. In [49], the authors aim at automatically assessing mobile applications created with App Inventor, focusing on their *visual design*. In [50], the author develops a semi-automated assessment approach to measure whether learners understand how to design and implement a *database schema* in SQL. In this latter work, the focus is not on SQL programming but on the ability to conceive and implement a schema from a description given in natural language. In [51,52], the authors present a tool to automatically analyse

Scratch or Snap! projects. The focus is not on the submitted programs, but on whether the projects realised by learners contribute to develop their *computational thinking skills*. Another example, presented in [53,54], is the evaluation of *OpenGL assignments*, for which the goal is to check whether the submitted programs build the correct 3D scene. A last example is the integration of an automated code assessment feature in visualisation and simulation tools that are typically used for data structure and algorithm-related assignments [55,56].

Several systems have been developed not to directly assess a code but to measure how well learners cover them with tests. In [57], the authors propose to assess if test coverage is good with respect to test cases defined by learners. In [58], the author insists that test-driven development is important and proposes to use automatically graded test-first assignments to improve student performances. In [59], the platform proposed by the authors checks whether learners have written test cases to cover all the requirements of the assignments.

Assessed aspects can also be specific to a given programming paradigm, such as object-oriented programming [60,61]. In [60], the authors develop a violation manager component for Eclipse to assess whether a Java program contains possible violations of an object-oriented paradigm. In [62], the authors develop a system to provide feedback on flaws in Java programs for not novice programmers, such as the misuse of a visibility modifier or fields that should have been local variables, for example. Another aspect that is not directly related to code and that can be assessed is the understanding of query languages to work with databases [63,64].

Finally, automated assessment systems can also be used to grade computer-science-related artefacts that can be represented with code. In [65,66], the authors propose to automatically grade Entity-Relation diagrams, either from a raster-based image or from an XML representation of the diagrams. In [49], the visual design of mobile applications is assessed based on its App Inventor source code, which is a structured file. In [67], flowcharts can be assessed thanks to a conversion into BASIC programs that can undergo a set of tests.

### 2.2. *Methods and Techniques*

Techniques used to automatically assess codes can generally be classified in two main categories. On one side, *static* approaches are using algorithms developed for compilers and for language-based tools to analyse the text of the source codes without executing them [13]. These approaches include semantic similarity and graph-based techniques. On the other side, *dynamic* approaches analyse the output results after possibly compiling and then running the code with a set of predefined tests [1]. These approaches include test-based techniques, with tests generally either run following a black-box or a white-box approach. Finally, *hybrid* approaches combine both static and dynamic techniques [68,69].

#### 2.2.1. Static Approaches

Static approaches are based on any tool or algorithm that can analyse a source code, starting with compilers and interpreters. Other existing tools that can be used include static source code analysers (CheckStyle, PMD, etc.) and model checkers (Java PathFinder, ESC/Java, etc.). Analysis techniques using a static approach are generally used to assess code syntax, anti-plagiarism and code quality aspects. Three main categories of static analysers' features can be identified: detecting fault or errors, checking the format and measuring properties [45]. In [26], the authors propose to use static analysis to identify whether concepts specified by instructors are present in learners' submissions. Their analysis is based on key abstractions and keyword extraction from submitted codes. In [20], regular expressions are used to check for specific syntactic errors learners are commonly doing. In [62], the abstract syntax tree is used to perform data flow analyses to identify flaws in Java programs.

It is also possible to compute several kinds of metrics from source codes. Such measures are generally used to assess code quality aspects [70] but are also used to detect plagiarism when computed on pairs of codes. In [71], two metrics are used to assess the structural complexity of submitted codes. In [46], the metrics to consider can be selected from a list of available ones, and measurement value limits along with weight can be configured by instructors. In [72], the authors are computing metrics on graph representations of source codes to assess structural aspects and compare codes. In [73], the authors are building dependence graphs of submitted codes and comparing them to a model program to take into account that several solutions are possible for a given problem specification. In [44], the authors are using a similarity metric on submissions to build a chain of codes, each one having an additional specific language concept or idiom. This data structure is built from a big corpus of submissions and used to provide actionable style hints to learners. In [65], similarity between submissions and the instructor's solution is measured thanks to the computation of edit distances between them.

### 2.2.2. Dynamic Approaches

Dynamic approaches gather techniques requiring the execution of the code. They are usually based on an analysis of the output produced by the code execution, comparing them with those of a reference solution. In this case, they are usually referred to as unit testing approaches and are used to assess code semantic aspects. Industrial tools such as JUnit for Java programs [33,61,74] are usually used for such approaches. As highlighted in [75], working with test cases requires instructors to define limiting or abnormal cases to train students to produce bullet-proof software. In [76], the author develops a framework and systematic approach to design tests that can be used for automated assessment, mainly focusing on precise specifications with a well-defined perimeter on what to test and how to test it. The analyses can be performed on string representations of the output results, making them programming language-agnostic. However, to obtain more precise and detailed assessments, the analyses of the output results must be directly performed on the domain values. Some systems support both kinds of analyses of the output results. In [77,78], the authors use a dynamic unit testing-based approach that can both compare string representations or domain values of output results. The second kind of analysis is only available for a few programming languages because libraries have not been written for many of them for yet.

In addition to just comparing the produced and expected outputs, using a string or exact value comparison, some approaches perform additional analyses on the produced output. In [79], the authors propose to perform a semantic-similarity analysis on the output results. This approach makes it possible to compare the meaning of the produced output with the expected output. It also opens the possibility for partial grading since it makes it possible to distinguish a nearly correct answer. For example, the submitted code does produce the correct values in an array, but they are not sorted in the expected order. In [80], a simpler approach based on regular expressions is used to identify the output that have to be considered as correct. Other test-based approaches are also possible. In [71], the authors are using property-based black-box testing to develop test models against which submitted codes are run. From the obtained results, their system ranks the submissions according to the amount of contained bugs and infers the algorithmic complexity.

Dynamic approaches can also benefit from features that are specific to the programming language. In [80], specific features of object-oriented programming languages are used to build more a genuine tester. Reflection and inheritance are indeed used to better analyse learners' individual submissions without imposing too many constraints on their code. In [61], Java interfaces are used to define a common API that all submitted codes must respect, making the use of reflection for the assessment system easier. In [34], the author proposes a technique to assess the design of GUI in Java by using a custom library that exists in two versions: one that can be used to build user interface and one that is a mock used for the assessment.

Finally, dynamic approaches can also be used to assess real code performance aspects in a controlled execution environment. These kinds of analyses are heavily used in automated code assessment systems used for contests. When used for education, performances measures are generally used to enforce time or memory constraints for the assignments and for security reasons.

### 2.2.3. Hybrid Approaches

Hybrid approaches combine both static and dynamic analyses to produce a code assessment on one given aspect. Such approaches are usually considered when assessing programs that are producing code as output. In [36], the execution of submitted codes produces SVG charts that are then analysed with a static approach to extract the information needed to assess the submission. In [81], the authors propose to assess web applications by checking that the produced web applications contain the expected component. For that, the code of the generated HTML pages is examined using static analyses. Hybrid approaches combining static and dynamic analyses are also used to make it possible for code submissions containing syntactic errors or not producing any input to be graded with anything but a zero mark [73]. This is very important in the particular case of courses targeted to novice programmers.

Another possible issue with automated code assessment systems is that a non-compiling program is usually associated with a zero mark since it is not possible to run tests against a code that cannot be compiled. To tackle this problem, hybrid approaches first trying to repair the program with algorithms based on static analysis techniques before running tests are being developed [82,83]. Using such approaches makes it possible to build an assessment resulting in a partial grading.

### 2.2.4. Modelling

Other approaches are not performing analyses directly on submitted codes but on models representing them. In [84], the authors propose to use symbolic execution to detect possible path differences between submitted codes and a reference implementation solution. This makes it possible to assess whether submitted codes implement all the same behaviours as the reference one.

### 2.2.5. Artificial Intelligence

Artificial intelligence is also being used to automatically assess different aspects of computer programs. In [85,86], *machine learning* techniques are used to grade programs based on a grammar of features capturing signature elements that human experts are looking at when assessing a code. In [65], machine learning is also used but to assess whether ER diagrams are close to the expected solution. In [48], the authors use a convolutional *neural network* to measure the robustness of a web page, comparing screenshots of the page before and after the addition of new content. The result produced by the neural network is combined with other metrics to obtain a single robustness score. In [87], *clustering* is used to automate style grading by identifying common mistakes. The authors make the hypothesis that there might only be a limited number of ways to solve a problem.

Table 1 summarises the main methods and techniques reviewed in this section, organised by categories and related to the code aspect they are used to assess. Code performance is not in the table since no specific techniques to assess this aspect have been found.

**Table 1.** Different methods and techniques can be used by automated code assessment systems to assess four of the five identified code and program aspects.

| **Method/Technique** | **Code Syntax** | **Anti-Plagiarism** | **Code Semantic** | **Code Quality** |
|---|---|---|---|---|
| Static approach | • AST analysis [40,62]<br>• Keyword search [21,37]<br>• Regular expression [20]<br>• Dependence graph similarity [73] | • AST comparison [21] | • Data type checking [37]<br>• Edit distance [65] | • AST analysis [21,40,42,62]<br>• Metrics [40,45,57]<br>• Edit distance [44] |
| Dynamic approach | | | • Test-based [21,35,38,47,75,76,88,89]<br>• Unit-testing [33,40,61]<br>• Object-oriented programming features [80]<br>• Mock testing [34]<br>• Semantic similarity [79]<br>• Property-based testing [71] | • Interaction test [47] |
| Hybrid approach | | | • Test-based and element extraction [36,81] | |
| Modelling | | | • Symbolic execution [84] | |
| Artificial Intelligence | | | • Machine learning [65,85] | • Machine learning [85]<br>• Neural network [48]<br>• Clustering [87] |

### *2.3. Feedback*

Automated code assessment systems are not only used to support instructors for grading duties but also to accompany learners. Giving learners feedback is an effective way to promote learning [90], especially in the case of formative ones [91]. The quality of feedback produced by the automatic assessment process is therefore crucial. Feedback is indeed important when learning programming, in particular with educational games, as they provide information about the learning process to both the instructors and learners [92].

A challenge with automatically generated feedback is that they can feel mechanical and inauthentic. This can result in mindless repetitive attempts from learners to second guess the mechanism, which interfere with an efficient learning process. Generally, feedback consists of marks used for summative assessment and textual comments for formative assessment.

#### 2.3.1. Status

The simplest feedback for an assessment is a pass-or-fail status. Such a very limited feedback is, of course, not so useful in an educational context. Instead of only two different possible statuses, automated code assessment systems usually have more of them. In [38], the authors propose to associate one of four status to each test case: "accepted" for a successful test, "format error" if the produced output is correct but not well-formatted, "wrong answer" if the produced output is incorrect and "time limit exceeded" if the submitted code did not finish its execution within the time limit defined by instructors. In [37], five different values can be produced as a feedback, depending on the results of the assessment of a program written in assembly language. The submission can fail to compile, can take too much time to execute, can produce wrong results for some tests, can miss some quality requirements or finally can be "perfect". In [33], four statuses corresponding to the level of achievement are used: "excellent", "good", "satisfactory" and "poor". In [65], five statuses are used to characterise the closeness to the expected solution of submitted ER diagrams.

#### 2.3.2. Mark

Since automated code assessment systems are usually used for summative assessment, it is important for instructors to be able to obtain a mark for each submission. In [26], a grade for a submitted code is produced by summing up all the individual obtained grades resulting from the presence of key abstraction or keywords specified by instructors, resulting in one overall score. For test-based approaches, the mark can be as simple as the number of tests that succeeded over the total number of tests [80].

In [93], the authors define a *feedback unit* as the response awarded to learners to help them measure their progress towards achieving a "successful" program. The generated response are grades that can have a coarser or finer granularity, associated with textual feedback identifying tests that passed and, when possible, possible places to look at in the submitted codes for improvement. Learners tend to perform differently depending on the granularity used for the marks and associated feedback. Proposing more detailed marks encourages learners to make progress and be active. In addition, for such marks to be of interest and useful for learners, fine-grained tests should be used [94]. In [95], points are also awarded for each successful test, but penalties are then removed, for example, if the thrown exception choice was wrong. In [58], two scores are produced to measure how well a code has been tested. A validity score measures the accuracy of the tests and a completeness score measures the test coverage.

Marks produced by an automated code assessment platform can result from the combination of several marks, each of them assessing a specific aspect [71,72,83,96]. Weights are often used to indicate the importance of each assessed aspect. In these situations, the detailed scoring scheme is usually communicated to learners before the assessment.

### 2.3.3. Rubric

Using rubrics for an evaluation is a common practice because it makes evaluation criteria, and therefore learning goals, explicit for both learners and instructors [97]. Rubrics are mainly found for the assessment of quality aspects. In [98], the authors propose a rubric that can be used to provide feedback for code quality aspect that should be taught in programming courses. In [99], the author proposes a rubric with two parts, one to assess general style and design issues and one specific to the problem to be solved. In [100], the authors develop a tool with modifiable grade rubrics. In [72], the authors propose that instructors select assessment criteria to use when creating an assignment from a set of predefined ones organised into five categories. In [49], the authors propose a rubric to assess mobile application implemented with App Inventor. The latter can be used to support an automatic assessment of the visual design of the submitted programs. In [52], the authors also design rubrics to assess whether App Inventor and Snap! projects contribute to develop computational thinking skills. In [86], the authors propose a generic rubric that is linked with learners' ability to develop an algorithm for a given problem.

### 2.3.4. Counterexample

Assessment systems that are using test-based approaches usually include counterexamples in their feedback. The majority of tools include failed test cases in the generated feedback [25,71,94]. These are important to explain to learners why their codes are unsatisfactory, help them to debug the code they submitted and also avoid most cases of learners protesting. In [84], symbolic executions of submitted codes and a reference solution are compared, and discovered path deviations are used to identify semantic differences. The latter result in counterexamples as feedback, helping learners to understand what their codes are missing to fix them.

In [47], the authors present a platform used to automatically assess web applications. The feedback report generated by the tool contains step-by-step animation playbacks for each evaluation that has been run for the requirements, with informative textual feedback. These feedback animations consist of screenshots sequences of the browser contents, allowing learners to repeat the same scenario to correct their submissions.

### 2.3.5. Comment

Feedback can provide different kinds of comments to learners about the results of the assessments that have been carried out on their code submissions. It is also important to provide relevant comments when the assessment is not completely successful [21]. The simplest possible comments consist of the error message if the code does not compile or produces an execution errors or details about the tests that failed. In addition, it is important for the comments to be relevant and constructive and to include warnings and encouragements.

Several techniques have been developed to produce feedback comments that are as relevant as possible. In [101,102], the authors develop a technique to improve the compilation error messages, making them more understandable by learners and helping them to correct their code submissions. Such "translations" of errors usually detected by compilers or static analysers are very important in the case of novice programmers. In [59], execution errors and uncaught exceptions are also rephrased to help learners identify the root cause of their issues. In [83], the error type and the compiler message are combined to generate a useful feedback message enriched with valid and invalid statement examples, when relevant. In [103], the authors propose using clustering techniques to gather similar code submissions into clusters in order to identify error classes. The idea is then to associate specific and relevant feedback messages for each cluster, explaining what is the corresponding error class. In [104], the authors propose to automatically generate feedback by comparing the submitted code with a reference implementation. The produced comments are meant to direct learners towards the reference implementation. In [44], the authors propose an approach that generates hints about how to improve the style of submitted codes. Trying to indicate to learners what the causes of the reported errors are is important for a comment to be effective [25]. This latter observation explains why several tools make it possible for instructors to manually add comments. In [100], the authors propose a system in which instructors can annotate the submissions with custom comments and motivational stickers. In [67], hints are also provided to learners, but with an objective to explain to them how they can improve their submissions.

#### 2.3.6. Report

Finally, feedback can take the form of a report containing various information for learners, possibly combining marks and comments explaining them. In [47], a report with a summary of the whole assessment is produced along with detailed feedback for each requirement of the assignment and the corresponding test cases. In [40], the authors present a tool which generates reports based on a combination of the output produced by several industrial JavaScript analysis tools. In [88], Juedes presents a tool to evaluate programming projects which generates feedback in the form of a tree of web pages. The produced feedback takes the form of a report indicating whether the submitted programs compile, pass a set of tests, are correctly designed and are well documented. Each report entry contains links to detailed information such as the shell output and the produced and expected results for test cases [105]. In [106], the authors develop a technique to assess whether chosen learning outcomes have been acquired by learners. To assess this, metrics are computed on submitted codes and a probability that these programs contain convincing evidences of the mastery of the learning objectives is derived. Finally, the generated report contains a translation of these probabilities into textual feedback.

#### 2.3.7. Other Kinds of Feedback

Other kinds of feedback are also possible. In [32], the authors propose to include a screenshot of the mobile application just before a failure is detected, to help learners debug their code. This screenshot can also be used for learners to interpret the test results and the possible error stack trace.

Table 2 summarises the main kinds of produced feedback reviewed in this section, organised by categories and related to the code aspect they are used to assess. Anti-plagiarism is not in the table since no specific feedback reporting about this aspect has been found.

**Table 2.** Different kinds of feedback can be generated for instructors and learners for each kind of aspects that can be assessed on a submitted code or program.

| **Method/Technique** | **Code Syntax** | **Code Semantic** | **Code Performance** | **Code Quality** |
|---|---|---|---|---|
| Status | • Accepted or not [38] | • Test-output associated [38]<br>• Level of achievement [33]<br>• Closeness to expected solution [65] | • Constraint satisfied or not [38] | • Level of achievement [33] |
| Mark | | • Test-output associated [21]<br>• Weighted test cases [35]<br>• Proportion of succeeded tests [80]<br>• Composite grade [71,72,95,96] | | • Metrics values associated [72]<br>• Test validity and completeness [58] |
| Rubric | • Computational thinking associated [52] | • Ability to design algorithm [86] | | • Generic rubric [98,99]<br>• Visual design associated [49] |
| Counterexample | | • Failed unit tests [40,71,94]<br>• Execution path [84] | | • Failed interaction test [47] |
| Comment | • Wrong code patterns [20]<br>• Syntax error translation [59,83,101,102] | • Test-output associated [21,89]<br>• Execution error and exception translation [59] | • Complexity associated [40] | • Hint [44] |
| Report | | • Grading log [76]<br>• Test cases log [35,40,47,61] | | • Metrics log [40]<br>• Style analyses log [42] |

## 3. Automated Assessment Tools

Since the beginning of the development of automated code assessment techniques, tools to support them have been developed. As detailed in [1], they evolved from scripts to web-oriented systems, going through tool-oriented applications. No tool seems to dominate the market, and many tools are still being developed nowadays. Currently developed systems are ranging from specific ones developed for one particular context to more generic ones trying to integrate several existing tools into coherent systems.

### *3.1. Features*

Four main features of automated code assessment systems are highlighted in [107]: organising the assignments, receiving and storing learners' submissions, supporting automatic or semi-automatic assessment and providing feedback.

In addition, the possibility to integrate the developed tools with *learning management systems* is considered as an essential feature by many researchers. For example, the GitGrade system is integrated with Canvas [100]. Such an integration generally makes it possible to directly have support for the two first aforementioned features, namely, the assignments and submissions management.

Integrating *plagiarism detection* tools are also an important feature requested by instructors. Automated code assessment systems should generally be able to detect possible plagiarism among code submissions for an assignment in a course. Depending on the context, other sources may be used for the comparisons, such as code submissions from previous semesters for the same assignment or from other courses or even codes available on the Internet. Again, using learning management systems may make it easier to manage a repository of all the submissions for a given assignment.

### *3.2. Tools and Systems*

Table 3 summarises the tools analysed in the research presented in this paper. For each of them, it provides the key references and some information on the kind of tool, the code and program aspects that are assessed, the methods and techniques used for that, the generated feedback, the supported programming languages and indications on whether it integrates anti-plagiarism checks or not.

Most developed tools are web-based platforms that perform several analyses on submitted codes: checking syntax, testing for plagiarism, running test cases and evaluating some code quality aspects. *CodeMaster* is dedicated to the assessment of block-based programs for mobile applications, measuring their complexity with respect to computational thinking dimensions. It is highly configurable and allows instructors to select the analysis to perform. [52,67]. The *js-assess* tool is a client-side web application relying on several industrial tools to assess JavaScript programs [40]. *ProgEdu* is specific to Java programs with a focus on code style violations [42,120]. *WBGP* is a web-based TCL/TK software that makes it possible to annotate submitted codes with comments defined in a structured way [88,105]. The *BOSS* system has both an application and a web-based client. It supports summative and formative assessments since learners can run tests before the final submission and instructors can have their own test set for the final grading [109]. The *ProtoAPOGEE* system can be used to test for semantic and for some quality aspects such as robustness, resulting in a summary report with counterexamples for both aspects [47]. The *E-Lab* tool automatically generates test cases from a reference solution provided by instructors and then runs them against codes submitted by learners.

**Table 3.** Automated code assessment tools and systems can be characterised by several aspects summarised in this table: the kind of system, the code and program aspects that can be assessed with the methods and techniques used for the assessment, the kind of supported feedback, the supported programming languages and the anti-plagiarism tool used. A dash (–) in an entry means that the information has not been found in any reference.

| Tool | References | Kind of Tool or System | Code and Program Aspects | Methods and Techniques | Feedback | Programming Languages | Anti-Plagiarism |
|---|---|---|---|---|---|---|---|
| ACCE | [87] | Script | Quality (style) | AST edit distance and clustering | – | Python | – |
| Apollo | [106] | Program | Learning objective | Static analyses and metrics (PMD) | Report | Processing | – |
| ArTEMiS | [89] | Web-based | Semantic | Unit-testing | Report | Agnostic | – |
| AutoGrader | [61] | Program | Semantic and quality | Unit-testing and metrics (PMD) | Report | Mainly Java | – |
| AutoGrader | [84] | Python program | Semantic | Symbolic execution | Counterexample | – | – |
| AutoLEP | [73] | Program | Syntax and semantic | Dependence graph similarity and test-based | Counterexample and report | – | – |
| Automata | [85,86] | Web-based | Semantic and quality | Test-based and machine learning | Mark and report | – | – |
| AutoStyle | [44,108] | GUI program | Quality (style) | Edit distance | Status and comment | – | – |
| AWAT | [81] | Program | Semantic | Test-based (web-browser automation) | – | Agnostic | – |
| BOSS | [109] | Program and Web-based | Semantic and quality | Unit-testing and metrics | Mark and report | Mainly Java | Sherlock |
| CAC++ | [110] | C++ library | Syntax | Static analyses | Comment | C, C++ | |
| CodeMaster | [49,52] | Web-based | Syntax | Keyword search | Mark, badge and rubric | App Inventor and Snap! projects | – |
| CodeOcean | [59] | Web-based | Syntax, semantic and quality | Unit-testing and linter | Comment | Agnostic | – |
| CourseMaster | [67,111] | – | Syntax, semantic and quality | Static analyses and test-based | Mark and comment | Java and C++ | – |
| eGrader | [72] | Program | Syntax, semantic and quality | Unit-testing, graph similarity and metrics | Mark, comment and report | Java | – |
| E-Lab | [112] | Web-based | Semantic | Test-based | – | C, C++ and Java | MOSS |
| Fitchfork | [11] | Web-based | Semantic and performance | Unit-testing | Comment and report | Agnostic | – |
| FrenchPress | [62] | Eclipse plug-in | Syntax and quality | Static analyses | Comment | Java | – |

**Table 3.** *Cont.*

| Tool | References | Kind of Tool or System | Code and Program Aspects | Methods and Techniques | Feedback | Programming Languages | Anti-Plagiarism |
|---|---|---|---|---|---|---|---|
| GAME | [96,113] | GUI program | Syntax, semantic and quality | Static analyses, input–output and metrics | Mark and report | Java, C and C++ | No |
| GitGrade | [100] | Web-based | Syntax, semantic and quality | Script and manual review | Mark, rubrics, comment | Agnostic | MOSS |
| Gradeer | [114] | CLI program | Syntax, semantic and quality | Script, unit testing, style analyses and manual review | Mark and comment | Java | – |
| GradeIT | [83] | Prutor plug-in | Semantic | Program repair and unit testing | Mark and comment | C | – |
| GRASP | [95] | Program | Semantic | Unit-testing | Mark | .NET framework languages | – |
| GUI_Grader | [35] | Backend program | Semantic | Test-based and user interaction simulation | Mark and report | Java | – |
| HoGG | [80,115] | Program | Semantic and quality | Test-based | Report | Java | – |
| Infandango | [94] | Web-based | Semantic | Unit-testing | Mark and counterexample | Java | – |
| INGInious | [116] | Backend server | Semantic | Unit-testing | Pass-or-fail and comment | Agnostic | – |
| JavAssess | [117,118] | Java library | Syntax and semantic | AST analyses and test-based | Mark | Java | |
| JEWL | [34] | Java library | Semantic | Unit-testing | – | Java | – |
| js-assess | [40] | Web-based (client-only) | Syntax, semantic, quality and performance | Unit testing, linter, style analyses and metrics | Report | JavaScript | – |
| mark44 | [119] | UNIX shell script | Syntax, semantic and quality | Input-output, metrics and manual review | Mark and report | C | – |
| OCETJ | [74] | Web-based | Semantic | Unit-testing | Comment | Java | – |
| Online Judge | [38] | Program | Semantic, performance and quality | Test-based | Status and report | – | Yes |
| ProgEdu | [42,120] | Web-based | Syntax, semantic and quality | Style analyses (CheckStyle) | Report | Java | Yes |
| ProtoAPOGEE | [47] | Web-based CMS | Semantic and quality | Test-based (web-browser automation) | Mark, report and counterexample | Agnostic | – |

**Table 3.** *Cont.*

| Tool | References | Kind of Tool or System | Code and Program Aspects | Methods and Techniques | Feedback | Programming Languages | Anti-Plagiarism |
|---|---|---|---|---|---|---|---|
| Pythia | [39,77,78] | Backend server | Syntax, semantic and quality | Input-output, unit testing and script | Mark, comment and report | Agnostic | – |
| PSGE | [76] | Unix-based program | Semantic | Test-based | Report and manual | Agnostic | – |
| Quiver | [75] | Web-based and GUI program | Syntax and semantic | Unit-testing | Comment | C++, Java and MIPS Assembly Language | – |
| SCAGrader | [45] | Web-based | Syntax | Style analyses | Mark | C, Java and PHP | – |
| Scheme-Robo | [21] | Email-based | Syntax and semantic | Structure analyses, input–output (value) and unit testing (fixed and random) | Mark and comment | Scheme | Yes |
| SPT | [60] | Eclipse plug-in | Quality | – | Report | Java | – |
| STAGE | [57] | Moodle plugin | Semantic and quality | Metrics | Mark | Java | – |
| Style++ | [46] | Program (CLI and GUI) | Quality | Style analyses | Mark and comment | C++ | – |
| VPL | [107] | Moodle plugin | Dynamic | Input-output, unit testing, style analyses and test coverage | Mark, comment and report | Agnostic | |
| WBGP | [88,105] | Software | Syntax, semantic and quality | Script and manual review | Mark, comment and report | Agnostic | MOSS |
| WebBot | [25,121] | Web-Based | Semantic | Input-output | Counterexample and comment | Multi-language | History comparison |
| Web-CAT | [58,122] | Program | Semantic and quality | Unit-testing and test coverage | Mark and report | Agnostic | – |

Several existing tools only have an application client. The *AutoStyle* system is a GUI program that can be used by instructors and learners to assess code style and provide three kinds of hints: approach, syntactic and code skeletons [44,108]. The *Scheme-robo* tool is even simpler, as learners just submit their code by sending an e-mail to a specific address. They then receive a reply with a copy of their submission and a set of points with comments for it [21]. The *GAME* tool is a GUI program that grades Java, C and C++ projects based on a marking schema and strategy defined by instructors [96]. The *Apollo* program is a tool to measure whether learners master learning objectives by extracting evidences from submitted codes for assignments that have been specifically designed for each learning objective. The evidences are obtained from metrics computed with PMD [106]. *AutoGrader* runs symbolic executions of submitted codes and compares them with a reference solution. This approach avoids the need to define or generate test cases while still being able to provide counterexamples to learners when their submissions are wrong [84]. *Web-CAT* is famous for grading how well a program has been tested by learners and used in many courses that aim at encouraging learners to write tests [58,122]. The *Gradeer* tool is a simple CLI program that is working in the other way round, in the sense that it is meant to be used to assist a manual assessment [114].

Many tools are custom tailored solutions for specific programming languages generally relying on existing algorithms and libraries to perform analyses. The *ArTEMiS* tool follows an approach based on continuous integration (CI) allowing learners to upload codes though a version control system to obtain immediate feedback from the CI server. It makes it possible for this tool to be language-agnostic and highly configurable by defining custom CI pipelines [89]. The *Gradeer* tool consists of scripts corresponding to checks that can be performed on a code, each resulting in a mark and a comment that are gathered to form a global feedback [114]. The *GRASP* tool is focused on the assessment of project written for the .NET framework and implementing a component by only using unit testing [95]. The *eGrader* tool only focuses on Java and provides outcome analyses for instructors, along with a database with statistics on all submissions [72].

Specific and simpler systems includes *Infandango*, an open source, web-based system that can run Java programs against a set of fine-grained JUnit unit tests to assign a grade to each submitted program [94]. The *mark44* system is also very simple, as it is implemented as a single UNIX shell script that can assess C programs to produce a mark from tests, metrics and human inputs [119]. The *PSGE* system is also a simple UNIX-based program that can handle fully and semi-automated assessments of code following a test-based approach [76]. The *ACCE* tool is just a set of scripts that computes the similarity of submitted codes with other submissions to perform a clustering to assess style aspects [87].

Several tools have been built in a highly customisable way by making it possible to combine them with external modules for more specific analyses. The *Style++* tool can integrate style analysers for C++ programs [46]. The *SCAGrader* tool uses industrial tools to assess the style of C, Java and PHP programs, and its grader server can be used by a dedicated GUI or as a service by other tools [45].

Finally, some tools are just libraries that can be used to develop custom course management systems or can be integrated within existing learning management systems. *JEWL* is a Java library to create GUI with a mock version that can be used for automatic assessment [34]. *CAC++* is a C++ library developed to help instructors design their own assessment programs for C and C++ assignments [110]. Some systems have also been implemented as programs to be run as backend services to ease the integration with existing frontend ones. The *GUI_Grader* program is used to assess whether a GUI matches a set of requirements defined with a database describing the constraints and requirements for each GUI component [35]. The *INGInious* tool has been develop to assess code submitted on the edX platform in the context of an MOOC [116]. The *Pythia* platform has also been implemented as a backend server whose features can be integrated in several frontends, in particular thanks to its REST API [39,77,78].

### 3.3. Security

An important aspect of automated code assessment systems is that special attention should be paid to computer security. They indeed are generally running codes submitted by learners, or at least analyses on them, which may be harmful, should it be voluntary or not. Generally, automated code assessment systems are executing code submissions inside *sandboxes* [11,94,112]. These are usually implemented as disposable virtual machines created for a specific job or as *containers* to provide lighter isolation as used in [116]. Other solutions can be used, such as *User-Mode Linux* as used in [39].

## 4. Integration in the Learning Process

Automated code assessment systems can be used for several purposes, combined with different pedagogical approaches and in various phases of the learning process. Moreover, some systems are only designed for summative or formative assessments, while the majority of them can be used for both kinds.

### 4.1. Grading

The initial and main use of automated code assessment systems was to help instructors to correct assignments and projects made by learners. With such systems, there are no interactions with learners, except for the submission of their works. A report is then produced for instructors, possibly with proposed marks for each submission. The benefit in the learning process is the possibility for learners to have more assignments to work on since the correction time is reduced for instructors. Quickly, the assessment systems evolved and became tools to support learners by encouraging them to work and helping them to learn.

### 4.2. Active Learning

Pedagogical approaches based on *active learning* are adapted to the use of automated code assessment systems that are able to produce feedback for learners. Several pieces of research highlight the fact that instant feedback contributes to helping learners to make progress incrementally and continuously and learn from their failures. In [93], the authors propose using automated code assessment at different stages of a project. The aim is to provide learners with feedback on work-in-progress submissions prior to the final submission. Pre-deadline results improve as the number of feedback units increase, and post-deadline activities also improve as more feedback units are available. Learners are definitely encouraged to work and make progress when feedback is available and precise.

*Instant feedback* allows learners to iteratively solve assignments. In [74], the authors propose to define a public and a private test suite so that the public one can be used by learners for self-evaluation while working on the assignments and the private one is used for marking purpose. Such a setting is usual for unit-testing-based approaches. In [42], the authors develop an iterative learning environment, allowing learners to submit codes until meeting the necessary requirements or reaching the assignment deadline. Their system also makes it possible to track changes between submissions. In [89], the same possibility to iteratively solve exercises with resubmissions is used to make active learning possible in the context of large classes. In [46], both learners and instructors are using an automated code style assessment tool. Learners use it to obtain feedback and adjust their code while working on it. Instructors use it for the evaluation of the final version submitted for the given deadline.

### 4.3. Learning Behaviour

Another possible use of automated code assessment systems is the monitoring of learners' behaviour. Information collected by these systems can help instructors to better understand learners and how they are performing. In [123], the authors analyse data collected by an automated assessment system with data mining techniques. They manage to identify patterns that are predictive of final achievements for the course.

### *4.4. Semi-Automated Code Assessment*

Only relying on automated code assessment systems may not be enough to support learners in their learning process, and there should generally be a possibility for a human assessment as well. In [119], the author develops a simple semi-automated code assessment system based on a UNIX shell script, which can prompt human instructors for more advanced judgements. More advanced tools have also been designed to let human experts perform the grading, with a significant help of software support. For example, WBGP and GitGrade both make it possible for instructors to browse through code submissions, establish a mark and annotate them with feedback comments [88,100].

## **5. Discussions**

This section discusses whether the review presented in this paper manages to bring answers to the three addressed research questions.

### *5.1. Feasible Kinds of Assessment*

The review conducted for the research presented in this paper highlights several considerations related to automated code assessment systems. It makes it possible to identify several possible kinds of assessments that can be performed on single code or more complete program or project submissions. Usually, code assessment tools are focusing on syntax and semantic aspects, which is typically limited to test-based analyses. But depending on the learning objectives instructors want for their learners, it is also possible to put a focus on plagiarism checks and on code performance and code quality assessment. These aspects presented in Section 2 provide answer elements to the first research question.

### *5.2. Assessment Quality*

The second research question is about the quality of the automatically produced assessments. Only a few studies have been conducted to compare automatic and manual code assessments. Even if they tend to show that automatic assessments can be as good as human grading for several criteria such as fairness and consistency, the proposed review is not conclusive for this question. In addition, the existence of semi-automatic systems where instructors are helped by automated code assessment techniques tends to show that human input is still needed in certain contexts, as highlighted in Section 4.

### *5.3. Challenges for Automated Code Assessment*

Finally, the third research question asks about the main challenges related to the development of automated code assessment systems. The first category of challenges that has been identified consists of technical ones. To obtain better assessment with more useful and relevant feedback, new methods and techniques must be designed. Only a few studies use artificial intelligence or big data techniques, which may be explained by the youth of the domain and the non-existence of large datasets. A challenge may be to provide more "intelligent" assessments based on artificial intelligence techniques. Another issues is code plagiarism, which is unfortunately quite common, in particular for simple and small assignments. A challenge is to find ways to automatically generate different assignments that can be automatically assessed so that learners cannot copy codes from other learners anymore. The condition to maintain evaluation fairness would be to generate assignments targeting similar skills and with similar difficulty levels.

#### 5.3.1. Human Intervention and Assessment Quality

Fully automated code assessments do have their drawbacks, but they are sometimes the only way to go, for example, with large classes. Even if they contribute a lot in helping both instructors and learners, future developments should probably be focused on approaches combining automated and human grading. In [124], automated and manual assessments have been fully integrated, manual interventions being required all along the automatic grading. Such an approach that is not limited to the usual sequential one with a clear split between automatic and manual phases should maybe be explored more in the future.

### 5.3.2. Cheating and Creativity

Programming usually relies on a trial-and-error approach to fix errors and so do learners. When facing an automated code assessment system, they may use it following the same approach, making submissions until managing to find a "correct" one, at least according to the system. This is particularly true with systems using test-based approaches with a limited number of test cases that can be guessed by learners. As highlighted in [125], moving learners towards a reflection-in-action approach is a challenge for which solutions have to be found.

Another challenge due to cheating techniques used by fraudulent learners is that they may interfere with code analysis techniques [126]. This possible interference should maybe be part of the solution, in particular with code style analysis techniques. If automated code assessment systems do assess quality aspects, code obfuscation is maybe not an option for fraudulent learners.

A related challenge is the fact that most of the existing automated code assessment systems may reduce learners' creativity. This is usually the case with test-based approaches that often have such specific specifications and requirements to make it possible to assess them automatically that possible learners' submissions are limited.

### 5.3.3. Collaboration and Interoperability

Another challenge related to automated code assessment systems is the lack of collaborations [127]. As highlighted by the review presented in this paper, a large number of platforms are being developed. These platforms are often not designed to make it possible to reuse analyses components in other ones, making collaboration difficult. Several reasons can explain this observation. First of all, institutions usually develop a platform trying to first satisfy their own specific needs should they be related to the programming languages taught to their students or to requirement specific to the taught courses. Another reason is maybe the lack of common models to grade assignments and the wide variety of possible assignments [12]. However, several existing platforms are heavily relying on industrial tools to perform the analyses [40], which may be part of a solution towards more collaboration and cooperation. In addition, in the review presented in [1], three generations of automated code assessment systems are described. A possible fourth one could be tools implemented as an API, as cloud services or following a Learning Tools Interoperability (LTI) interface, making it easier to integrate them in learning management systems [32,59].

## 6. Conclusions

To conclude, the review presented in this paper covers recent pieces of research related to automated code assessment tools and systems. The research questions addressed in this paper led to a classification of the recent tools along different axes: the code and program aspects that are assessed, the methods and techniques used by the tools and the kinds of generated feedback. The review also highlighted several ways to integrate automated code assessment tools in the learning process.

Regarding the three research questions covered in this paper, the conducted review managed to bring answers to the first and third questions. Unfortunately, it was not possible to answer the second question asking whether the results obtained by automated code assessment systems are as good as those obtained by human graders or not. No pieces of research specific to this have been found.

In addition to present an up-to-date review covering several aspects of automated code assessment techniques and tools, this article brings a discussion on many challenges to tackle for the community of researchers. Possible solution elements emerged from the discussions and may be interesting directions of work towards more collaboration and towards better modular systems that can be easily used to cover different needs. Future work to follow on the proposed review is the development of a website and a community of practices to help researchers to work together.

**Funding:** This research received no external funding.

**Conflicts of Interest:** The author declares no conflict of interest.

## References

1. Douce, C.; Livingstone, D.; Orwell, J. Automatic Test-Based Assessment of Programming: A Review. *J. Educ. Resour. Comput.* **2005**, *5*, 215–221. [CrossRef]
2. Ala-Mutka, K.M. A Survey of Automated Assessment Approaches for Programming Assignments. *Comput. Sci. Educ.* **2005**, *15*, 83–102. [CrossRef]
3. Lajis, A.; Baharudin, S.A.; Kadir, D.A.; Ralim, N.M.; Nasir, H.M.; Aziz, N.A. A Review of Techniques in Automatic Programming Assessment for Practical Skill Test. *J. Telecommun. Electron. Comput. Eng.* **2018**, *10*, 109–113.
4. Keuning, H.; Jeuring, J.; Heeren, B. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Trans. Comput. Educ.* **2018**, *19*, 1–43. [CrossRef]
5. Aldriye, H.; Alkhalaf, A.; Alkhalaf, M. Automated Grading Systems for Programming Assignments: A Literature Review. *Int. J. Adv. Comput. Sci. Appl.* **2019**, *10*, 215–221. [CrossRef]
6. Ismail, M.H.; Lakulu, M.M. A Critical Review on Recent Proposed Automated Programming Assessment Tool. *Turk. J. Comput. Math. Educ.* **2021**, *12*, 884–894.
7. Rahman, K.A.; Nordin, M.J. A Review on the Static Analysis Approach in the Automated Programming Assessment Systems. In Proceedings of the 2007 National Conference on Programming, Montreal, QC, Canada, 21–25 October 2007.
8. Liang, Y.; Liu, Q.; Xu, J.; Wang, D. The Recent Development of Automated Programming Assessment. In Proceedings of the 2009 International Conference on Computational Intelligence and Software Engineering, Wuhan, China, 11–13 December 2009.
9. Ihantola, P.; Ahoniemi, T.; Karavirta, V.; Seppälä, O. Review of Recent Systems for Automatic Assessment of Programming Assignments. In Proceedings of the 10th Koli Calling International Conference on Computing Education Research, Koli, Finland, 28–31 October 2010; ACM: New York, NY, USA, 2010; pp. 86–93.
10. Romli, R.; Sulaiman, S.; Zamli, K.Z. Automatic Programming Assessment and Test Data Generation: A Review on its Approaches. In Proceedings of the 2010 International Symposium on Information Technology, Kuala Lumpur, Malaysia, 15–17 June 2010; pp. 1186–1192.
11. Pieterse, V. Automated Assessment of Programming Assignments. In Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research, Arnhem, The Netherlands, 4–5 April 2013; ACM: New York, NY, USA, 2013; pp. 45–56.
12. Caiza, J.C.; Alamo, J.M.D. Programming Assignments Automatic Grading: Review of Tools and Implementations. In Proceedings of the 7th International Technology, Education and Development Conference, Valencia, Spain, 4–6 March 2013; pp. 5691–5700.
13. Striewe, M.; Goedicke, M. A Review of Static Analysis Approaches for Programming Exercises. In Proceedings of the 2014 International Computer Assisted Assessment Conference, Zeist, The Netherlands, 30 June–1 July 2014; Springer: Berlin/Heidelberg, Germany, 2014; pp. 100–113.
14. Staubitz, T.; Klement, H.; Renz, J.; Teusner, R.; Meinel, C. 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, Zhuhai, China, 10–12 December 2015; pp. 23–30.
15. Arifi, S.M.; Abdellah, I.N.; Zahi, A.; Benabbou, R. Automatic Program Assessment Using Static and Dynamic Analysis. In Proceedings of the Third World Conference on Complex Systems, Marrakech, Morocco, 23–25 November 2015.
16. Keuning, H.; Jeuring, J.; Heeren, B. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education, Arequipa, Peru, 11–13 July 2016; ACM: New York, NY, USA, 2016; pp. 41–46.
17. Souza, D.M.; Felizardo, K.R.; Barbosa, E.F. A Systematic Literature Review of Assessment Tools For Programming Assignments. In Proceedings of the IEEE 29th International Conference on Software Engineering Education and Training, Dallas, TX, USA, 5–6 April 2016; pp. 147–156.
18. Gupta, S.; Gupta, A. E-Assessment Tools for Programming Languages: A Review. In Proceedings of the First International Conference on Information Technology and Knowledge Management, New Delhi, India, 22–23 December 2017; pp. 65–70.
19. Hollingsworth, J. Automatic Graders for Programming Classes. *Commun. ACM* **1960**, *3*, 528–529. [CrossRef]
20. Hegarty-Kelly, E.; Mooney, A. Analysis of an Automatic Grading System Within First Year Computer Science Programming Modules. In Proceedings of the Computing Education Practice, Durham, UK, 7 January 2021; ACM: New York, NY, USA, 2021; pp. 17–20.
21. Saikkonen, R.; Malmi, L.; Korhonen, A. Fully Automatic Assessment of Programming Exercises. In Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education, Canterbury, UK, 25–27 June 2001; ACM: New York, NY, USA, 2001; pp. 133–136.
22. Albluwi, I. Plagiarism in Programming Assessments: A Systematic Review. *ACM Trans. Comput. Educ.* **2020**, *20*, 1–28. [CrossRef]

23. Novak, M. Review of Source-Code Plagiarism Detection in Academia. In Proceedings of the 39th International Convention on Information and Communication Technology, Electronics and Microelectronics, Opatija, Croatia, 30 May–3 June 2016; pp. 796–801.
24. Novak, M.; Joy, M.; Kermek, D. Source-Code Similarity Detection and Detection Tools Used in Academia: A Systematic Review. *ACM Trans. Comput. Educ.* **2019**, *19*, 1–37. [CrossRef]
25. Colton, D.; Fife, L.; Thompson, A. A Web-Based Automatic Program Grader. *Inf. Syst. Educ. J.* **2006**, *4*, 7.
26. Alhami, I.; Alsmadi, I. Automatic Code Homework Grading Based on Concept Extraction. *Int. J. Softw. Eng. Its Appl.* **2011**, *5*, 77–84.
27. Kurtukova, A.; Romanov, A.; Shelupanov, A. Source Code Authorship Identification Using Deep Neural Networks. *Symmetry* **2020**, *12*, 2044. [CrossRef]
28. Karnalim, O.; Kurniawati, G. Programming Style on Source Code Plagiarism and Collusion Detection. *Int. J. Comput.* **2020**, *19*, 27–38. [CrossRef]
29. Cosma, G.; Joy, M. Towards a Definition of Source-Code Plagiarism. *IEEE Trans. Educ.* **2008**, *51*, 195–200. [CrossRef]
30. Hamilton, M.; Tahaghoghi, S.; Walker, C. Educating Students about Plagiarism Avoidance—A Computer Science Perspective. In Proceedings of the 2004 International Conference on Computers in Education, Melbourne, Australia, 30 November 2004; pp. 1275–1284.
31. Pierce, J.; Zilles, C. Investigating Student Plagiarism Patterns and Correlations to Grades. In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, WA, USA, 8–11 March 2017; ACM: New York, NY, USA, 2017; pp. 471–476.
32. Bruzual, D.; Montoya Freire, M.L.; Di Francesco, M. Automated Assessment of Android Exercises with Cloud-native Technologies. In Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, Trondheim, Norway, 15–19 June 2020; ACM: New York, NY, USA, 2020; pp. 40–46.
33. Khalid, A. Automatic Assessment of Java Code. *Maldives Natl. J. Res.* **2013**, *1*, 7–32.
34. English, J. Automated Assessment of GUI Programs using JEWL. *SIGCSE Bull.* **2004**, *36*, 137–141. [CrossRef]
35. Feng, M.Y.; McAllister, A. A Tool for Automated GUI Program Grading. In Proceedings of the 36th ASEE/IEEE Frontiers in Education Conference, San Diego, CA, USA, 27–31 October 2006.
36. Hull, M.; Guerin, C.; Chen, J.; Routray, S.; Chau, D.H. Towards Automatic Grading of D3.js Visualizations. In Proceedings of the 2021 IEEE Visualization Conference, New Orleans, LA, USA, 24–29 October 2021.
37. Lingling, M.; Xiaojie, Q.; Zhihong, Z.; Gang, Z.; Ying, X. An Assessment Tool for Assembly Language Programming. In Proceedings of the 2008 International Conference on Computer Science and Software Engineering, Wuhan, China, 12–14 December 2008; pp. 882–884.
38. Cheang, B.; Kurnia, A.; Lim, A.; Oon, W.C. On Automated Grading of Programming Assignments in an Academic Institution. *Comput. Educ.* **2003**, *41*, 121–131. [CrossRef]
39. Combéfis, S.; le Clément de Saint-Marcq, V. Teaching Programming and Algorithm Design with Pythia, a Web-Based Learning Platform. *Olymp. Informat.* **2012**, *6*, 31–43.
40. Karavirta, V.; Ihantola, P. Automatic Assessment of JavaScript Exercises. In Proceedings of the 1st Educators' Day on Web Engineering Curricula, Vienna, Austria, 5–9 July 2010.
41. Keuning, H.; Heeren, B.; Jeuring, J. Code Quality Issues in Student Programs. In Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education, Bologna, Italy, 3–5 July 2017; ACM: New York, NY, USA, 2017; pp. 110–115.
42. Chen, H.M.; Chen, W.H.; Lee, C.C. An Automated Assessment System for Analysis of Coding Convention Violations in Java Programming Assignments. *J. Inf. Sci. Eng.* **2018**, *34*, 1203–1221.
43. Jansen, J.; Oprescu, A.; Bruntink, M. The Impact of Automated Code Quality Feedback in Programming Education. In Proceedings of the 2017 Seminar Series on Advanced Techniques and Tools for Software Evolution, Madrid, Spain, 7–9 June 2017.
44. Moghadam, J.B.; Choudhury, R.R.; Yin, H.; Fox, A. AutoStyle: Toward Coding Style Feedback at Scale. In Proceedings of the 2nd ACM Conference on Learning @ Scale, Vancouver, BC, Canada, 14–18 March 2015; ACM: New York, NY, USA, 2015; pp. 261–266.
45. Yulianto, S.V.; Liem, I. Automatic Grader for Programming Assignment Using Source Code Analyzer. In Proceedings of the 2014 International Conference on Data and Software Engineering, Bandung, Indonesia, 26–27 November 2014.
46. Ala-Mutka, K.; Uimonen, T.; Jarvinen, H.M. Supporting Students in C++ Programming Courses with Automatic Program Style Assessment. *J. Inf. Technol. Educ.* **2004**, *3*, 245–262. [CrossRef]
47. Fu, X.; Peltsverger, B.; Qian, K.; Tao, L.; Liu, J. APOGEE: Automated Project Grading and Instant Feedback System for Web-Based Computing. *ACM SIGCSE Bull.* **2008**, *40*, 77–81. [CrossRef]
48. Gradjanin, E.; Prazina, I.; Okanovic, V. Automatic Web Page Robustness Grading. In Proceedings of the 44th International Convention on Information, Communication and Electronic Technology, Opatija, Croatia, 27 September–1 October 2021.
49. Solecki, I.; Porto, J.; da Cruz Alves, N.; Gresse von Wangenheim, C.; Hauck, J.; Borgatto, A.F. 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, Portland, OR, USA, 11–14 March 2020; pp. 51–57.
50. Stanger, N. Semi-Automated Assessment of SQL Schemas via Database Unit Testing. In Proceedings of the 26th International Conference on Computers in Education, Manila, Philippines, 26–30 November 2018.

51. Moreno-León, J.; Robles, G.; Román-González, M. Dr. Scratch: Automatic Analysis of Scratch Projects to Assess and Foster Computational Thinking. *RED-Rev. Educ. Distancia* **2015**, *46*, 1–23.
52. von Wangenheim, C.G.; Hauck, J.C.; Demetrio, M.F.; Pelle, R.; da Cruz Alves, N.; Barbosa, H.; Azevedo, L.F. CodeMaster—Automatic Assessment and Grading of App Inventor and Snap! Programs. *Informat. Educ.* **2018**, *17*, 117–150. [CrossRef]
53. Hodgkinson, B.; Lutteroth, C.; Wünsche, B. glGetFeedback—Towards Automatic Feedback and Assessment for OpenGL 3D Modelling Assignments. In Proceedings of the 2016 International Conference on Image and Vision Computing New Zealand, Palmerston North, New Zealand, 21–22 November 2016.
54. Wünsche, B.C.; Chen, Z.; Shaw, L.; Suselo, T.; Leung, K.C.; Dimalen, D.; van der Mark, W.; Luxton-Reilly, A.; Lobb, R. Automatic Assessment of OpenGL Computer Graphics Assignments. In Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, Larnaca, Cyprus, 2–4 July 2018; ACM: New York, NY, USA, 2018; pp. 81–86.
55. Korhonen, A.; Malmi, L. Algorithm Simulation with Automatic Assessment. In Proceedings of the 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education, Helsinki, Finland, 11–13 July 2000; ACM: New York, NY, USA, 2000; pp. 160–163.
56. Kaila, E.; Rajala, T.; Laakso, M.J.; Salakoski, T. Automatic Assessment of Program Visualization Exercises. In Proceedings of the 8th International Conference on Computing Education Research, Koli, Finland, 13–16 November 2008; ACM: New York, NY, USA, 2008; pp. 101–104.
57. Pape, S.; Flake, J.; Beckmann, A.; Jürjens, J. STAGE: A Software Tool for Automatic Grading of Testing Exercises: A Case Study Paper. In Proceedings of the 38th International Conference on Software Engineering Companion, Austin, TX, USA, 14–22 May 2016; pp. 491–500.
58. Edwards, S.H. Improving Student Performance by Evaluating How Well Students Test Their Own Programs. *J. Educ. Resour. Comput.* **2003**, *3*. [CrossRef]
59. Serth, S.; Staubitz, T.; Teusner, R.; Meinel, C. CodeOcean and CodeHarbor: Auto-Grader and Code Repository. In Proceedings of the 7th SPLICE Workshop at SIGCSE 2021: CS Education Infrastructure for All III: From Ideas to Practice, online, 15–16 March 2021.
60. Ardimento, P.; Bernardi, M.L.; Cimitile, M. Towards automatic assessment of object-oriented programs. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings, Seoul, Korea, 5–11 October 2020; ACM: New York, NY, USA, 2020; pp. 276–277.
61. Helmick, M.T. Interface-Based Programming Assignments and Automatic Grading of Java Programs. In Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, Dundee, Scotland, UK, 25–27 June 2007; ACM: New York, NY, USA, 2012; pp. 63–67.
62. Blau, H.; Moss, J.E.B. FrenchPress Gives Students Automated Feedback on Java Program Flaws. In Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education, Vilnius, Lithuania, 4–8 July 2015; ACM: New York, NY, USA, 2015; pp. 15–20.
63. Abelló, A.; Burgués, X.; Casany, M.J.; Martín, C.; Quer, C.; Rodríguez, M.E.; Romero, O.; Urpí, T. A Software Tool for E-Assessment of Relational Database Skills. *Int. J. Eng. Educ.* **2016**, *32*, 1289–1312.
64. Chandra, B.; Chawda, B.; Kar, B.; Reddy, K.V.M.; Shah, S.; Sudarshan, S. Data Generation for Testing and Grading SQL Queries. *VLDB J.* **2015**, *24*, 731–755. [CrossRef]
65. Simanjuntak, H. Proposed Framework for Automatic Grading System of ER Diagram. In Proceedings of the 7th International Conference on Information Technology and Electrical Engineering, Chiang Mai, Thailand, 29–30 October 2015; pp. 141–146.
66. Thomas, P.; Waugh, K.; Smith, N. Experiments in the Automatic Marking of ER-Diagrams. In Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, Monte de Caparica, Portugal, 27–29 June 2005; ACM: New York, NY, USA, 2005; pp. 158–162.
67. Higgins, C.; Symeonidis, P.; Tsintsifas, A. The Marking System for CourseMaster. *ACM SIGCSE Bull.* **2002**, *34*, 46–50. [CrossRef]
68. Wang, Z.; Xu, L. Grading Programs Based on Hybrid Analysis. In Proceedings of the International Conference on Web Information Systems and Applications, Qingdao, China, 20–22 September 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 626–637.
69. Wang, J.; Zhao, Y.; Tang, Z.; Xing, Z. Combining Dynamic and Static Analysis for Automated Grading SQL Statements. *J. Netw. Intell.* **2020**, *5*, 179–190.
70. Breuker, D.M.; Derriks, J.; Brunekreef, J. Measuring Static Quality of Student Code. In Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education, Darmstadt, Germany, 27–29 June 2011; ACM: New York, NY, USA, 2011; pp. 13–17.
71. Earle, C.B.; Åke Fredlund, L.; Hughes, J. Automatic Grading of Programming Exercises using Property-Based Testing. In Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education, Arequipa, Peru, 11–13 July 2016; ACM: New York, NY, USA, 2016; pp. 47–52.
72. AlShamsi, F.; Elnagar, A. An Automated Assessment and Reporting Tool for Introductory Java Programs. In Proceedings of the 2011 International Conference on Innovations in Information Technology, Abu Dhabi, UAE, 25–27 April 2011; pp. 324–329.
73. Wang, T.; Su, X.; Ma, P.; Wang, Y.; Wang, K. Ability-Training-Oriented Automated Assessment in Introductory Programming Course. *Comput. Educ.* **2011**, *56*, 220–226. [CrossRef]

74. Tremblay, G.; Labonté, E. Semi-Automatic Marking of Java Programs Using JUnit. In Proceedings of the International Conference on Education and Information Systems: Technologies and Applications, Orlando, FL, USA, July 31–August 2 2003; pp. 42–47.
75. Ellsworth, C.C.; Fenwick, J.B.; Kurtz, B.L. The Quiver System. In Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, Norfolk, VA, USA, 3–7 March 2004; ACM: New York, NY, USA, 2004; pp. 205–209.
76. Jones, E.L. Grading Student Programs—A Software Testing Approach. *J. Comput. Sci. Coll.* **2001**, *16*, 185–192.
77. Combéfis, S.; de Moffarts, G. Automated Generation of Computer Graded Unit Testing-Based Programming Assessments for Education. In Proceedings of the 6th International Conference on Computer Science, Engineering and Information Technology, Zurich, Switzerland, 23–24 November 2019; pp. 91–100.
78. Combéfis, S.; Paques, A. Pythia reloaded: An Intelligent Unit Testing-Based Code Grader for Education. In Proceedings of the 1st Int'l Code Hunt Workshop on Educational Software Engineering, Baltimore, MD, USA, 14 July 2015; ACM: New York, NY, USA, 2015; pp. 5–8.
79. Fonte, D.; da Cruz, D.; Gançarski, A.L.; Henriques, P.R. A Flexible Dynamic System for Automatic Grading of Programming Exercises. In Proceedings of the 2nd Symposium on Languages, Applications and Technologies, Porto, Portugal, 20–21 June 2013; pp. 129–144.
80. Morris, D.S. Automatically Grading Java Programming Assignments via Reflection, Inheritance, and Regular Expressions. In Proceedings of the 32nd ASEE/IEEE Annual Frontiers in Education, Boston, MA, USA, 6–9 November 2002.
81. Sztipanovits, M.; Qian, K.; Fu, X. The Automated Web Application Testing (AWAT) System. In Proceedings of the 46th Annual Southeast Regional Conference on XX, Auburn, AL, USA, 28–29 March 2008; ACM: New York, NY, USA, 2008; pp. 88–93.
82. Gulwani, S.; Radiček, I.; Zuleger, F. Automated Clustering and Program Repair for Introductory Programming Assignments. *ACM Sigplan Not.* **2018**, *53*, 465–480. [CrossRef]
83. Parihar, S.; Dadachanji, Z.; Singh, P.K.; Das, R.; Karkare, A.; Bhattacharya, A. 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, Bologna, Italy, 3–5 July 2017; ACM: New York, NY, USA, 2017; pp. 92–97.
84. Liu, X.; Wang, S.; Wang, P.; Wu, D. Automatic Grading of Programming Assignments: An Approach Based on Formal Semantics. In Proceedings of the IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training, Montreal, QC, Canada, 25–31 May 2019; pp. 126–137.
85. Srikant, S.; Aggarwal, V. Automatic Grading of Computer Programs: A Machine Learning Approach. In Proceedings of the 12th International Conference on Machine Learning and Applications, Miami, FL, USA, 4–7 December 2013; pp. 85–92.
86. Srikant, S.; Aggarwal, V. A System to Grade Computer Programming Skills Using Machine Learning. In Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, New York, NY, USA, 24–27 August 2014; ACM: New York, NY, USA, 2014; pp. 1887–1896.
87. Rogers, S.; Tang, S.; Canny, J. ACCE: Automatic Coding Composition Evaluator. In Proceedings of the 1st ACM Conference on Learning @ Scale, Atlanta, GA, USA, 4–5 March 2014; ACM: New York, NY, USA, 2014; pp. 191–192.
88. Juedes, D.W. Web-Based Grading: Further Experiences and Student Attitudes. In Proceedings of the 35th ASEE/IEEE Frontiers in Education Conference, Indianopolis, IN, USA, 19–22 October 2005.
89. Krusche, S.; Seitz, A. ArTEMiS: An Automatic Assessment Management System for Interactive Learning. In Proceedings of the 49th ACM Technical Symposium on Computer Science Education, Baltimore, MD, USA, 21–24 February 2018; ACM: New York, NY, USA, 2018; pp. 284–289.
90. Hattie, J.; Timperley, H. The Power of Feedback. *Rev. Educ. Res.* **2007**, *77*, 81–112. [CrossRef]
91. Shute, V.J. Focus on Formative Feedback. *Rev. Educ. Res.* **2008**, *78*, 153–189. [CrossRef]
92. Combéfis, S.; Beresnevičius, G.; Dagienė, V. Learning Programming through Games and Contests: Overview, Characterisation and Discussion. *Olymp. Informat.* **2016**, *10*, 39–60. [CrossRef]
93. Falkner, N.; Vivian, R.; Piper, D.; Falkner, K. Increasing the Effectiveness of Automated Assessment by Increasing Marking Granularity and Feedback Units. In Proceedings of the 45th ACM Technical Symposium on Computer Science Education, Atlanta, GA, USA, 5–8 March 2014; ACM: New York, NY, USA, 2014; pp. 9–14.
94. Hull, M.; Powell, D.; Klein, E. Infandango: Automated Grading for Student Programming. In Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education, Darmstadt, Germany, 27–29 June 2011; ACM: New York, NY, USA, 2011; p. 330.
95. Cerioli, M.; Cinelli, P. GRASP: Grading and Rating ASsistant Professor. In Proceedings of the ACM-IFIP Informatics Education Europe III Conference, Venice, Italy, 4–5 December 2008; pp. 37–51.
96. Blumenstein, M.; Green, S.; Nguyen, A.; Muthukkumarasamy, V. GAME: A Generic Automated Marking Environment for Programming Assessment. In Proceedings of the 2004 International Conference on Information Technology: Coding and Computing, Las Vegas, NV, USA, 5–7 April 2004; pp. 212–216.
97. Allen, D.; Tanner, K. Rubrics: Tools for Making Learning Goals and Evaluation Criteria Explicit for Both Teachers and Learners. *CBE Life Sci. Educ.* **2006**, *5*, 197–203. [CrossRef]
98. Stegeman, M.; Barendsen, E.; Smetsers, S. Designing a Rubric for Feedback on Code Quality in Programming Courses. In Proceedings of the 16th Koli Calling International Conference on Computing Education Research, Koli, Finland, 24–27 November 2016; ACM: New York, NY, USA, 2016; pp. 160–164.

99. Becker, K. Grading Programming Assignments Using Rubrics. In Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education, Thessaloniki, Greece, 30 June–2 July 2003; ACM: New York, NY, USA, 2003; p. 253.
100. Zhang, J.K.; Lin, C.H.; Hovik, M.; Bricker, L.J. GitGrade: A Scalable Platform Improving Grading Experiences. In Proceedings of the 51st ACM Technical Symposium on Computer Science Education, Portland, OR, USA, 11–14 March 2020; ACM: New York, NY, USA, 2020; p. 1284.
101. Hristova, M.; Misra, A.; Rutter, M.; Mercuri, R. Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. *ACM SIGCSE Bull.* **2003**, *35*, 153–156. [CrossRef]
102. Blok, T.; Fehnker, A. Automated Program Analysis for Novice Programmers. In Proceedings of the 3rd International Conference on Higher Education Advances, Valencia, Spain, 21–23 June 2017.
103. Combéfis, S.; Schils, A. Automatic Programming Error Class Identification with Code Plagiarism-Based Clustering. In Proceedings of the 2nd Int'l Code Hunt Workshop on Educational Software Engineering, Seattle, WA, USA, 18 November 2016; ACM: New York, NY, USA, 2016; pp. 1–6.
104. Singh, R.; Gulwani, S.; Solar-Lezama, A. Automated Feedback Generation for Introductory Programming Assignments. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, Seattle, WA, USA, 16–19 June 2013; ACM: New York, NY, USA, 2013; pp. 15–26.
105. Juedes, D.W. Experiences in Web-Based Grading. In Proceedings of the 33rd ASEE/IEEE Frontiers in Education Conference, Westminster, CO, USA, 5–8 November 2003.
106. Rump, A.; Fehnker, A.; Mader, A. Automated Assessment of Learning Objectives in Programming Assignments. In Proceedings of the 17th International Conference on Intelligent Tutoring Systems, Athens, Greece, 7–11 June 2021; Springer: Berlin/Heidelberg, Germany, 2021; pp. 299–309.
107. del Pino, J.C.R.; Rubio-Royo, E.; Hernández-Figueroa, Z.J. A Virtual Programming Lab for Moodle with automatic assessment and anti-plagiarism features. In Proceedings of the 2012 International Conference on e-Learning, e-Business, Enterprise Information Systems, & e-Government, Las Vegas, NV, USA, 16–19 July 2012.
108. Choudhury, R.R.; Yin, H.; Moghadam, J.; Fox, A. AutoStyle: Toward Coding Style Feedback at Scale. In Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion, San Francisco, CA, USA, 26 February–2 March 2016; ACM: New York, NY, USA, 2016; pp. 21–24.
109. Joy, M.; Griffiths, N.; Boyatt, R. The BOSS Online Submission and Assessment System. *J. Educ. Resour. Comput.* **2005**, *5*, 2–es. [CrossRef]
110. Delgado-Pérez, P.; Medina-Bulo, I. Customisable and Scalable Automated Assessment of C/C++ programming assignments. *Comput. Appl. Eng. Educ.* **2020**, *28*, 1449–1466. [CrossRef]
111. Foxley, E.; Higgins, C.; Hegazy, T.; Symeonidis, P.; Tsintsifas, A. The CourseMaster CBA System: Improvements over Ceilidh. In Proceedings of the 5th International Computer Assisted Assessment Conference, Loughborough, UK, 2–3 July 2001.
112. Delev, T.; Gjorgjevikj, D. E-Lab: Web Based System for Automatic Assessment of Programming Problems. In Proceedings of the ICT Innovations 2012 Conference, Ohrid, North Macedonia, 12–15 September 2012; pp. 75–84.
113. Blumenstein, M.; Green, S.; Nguyen, A.; Muthukkumarasamy, V. An Experimental Analysis of GAME: A Generic Automated Marking Environment. *ACM SIGCSE Bull.* **2004**, *36*, 67–71. [CrossRef]
114. Clegg, B.; Villa-Uriol, M.C.; McMinn, P.; Fraser, G. Gradeer: An Open-Source Modular Hybrid Grader. In Proceedings of the 43rd ACM/IEEE International Conference on Software Engineering: Software Engineering Education and Training, Madrid, Spain, 25–28 May 2021.
115. Morris, D.S. Automatic Grading of Student's Programming Assignments: An Interactive Process and Suite of Programs. In Proceedings of the 33rd Annual Frontiers in Education, Westminster, CO, USA, 5–8 November 2003.
116. Derval, G.; Gego, A.; Reinbold, P.; Frantzen, B.; Roy, P.V. Automatic grading of programming exercises in a MOOC using the INGInious platform. In Proceedings of the European MOOC Stakeholder Summit 2015, Mons, Belgium, 18–20 May 2015; pp. 86–91.
117. Insa, D.; Silva, J. Semi-Automatic Assessment of Unrestrained Java Code: A Library, a DSL, and a Workbench to Assess Exams and Exercises. In Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education, Vilnius, Lithuania, 4–8 July 2015; ACM: New York, NY, USA, 2015; pp. 39–44.
118. Insa, D.; Silva, J. Automatic Assessment of Java Code. *Comput. Lang. Syst. Struct.* **2018**, *53*, 59–72. [CrossRef]
119. Jackson, D. A Semi-Automated Approach to Online Assessment. *ACM SIGCSE Bull.* **2000**, *32*, 164–167. [CrossRef]
120. Chen, H.M.; Chen, W.H.; Hsueh, N.L.; Lee, C.C.; Li, C.H. ProgEdu—An Automatic Assessment Platform for Programming Courses. In Proceedings of the 2017 International Conference on Applied System Innovation, Sapporo, Japan, 13–17 May 2017; pp. 173–176.
121. Colton, D.; Fife, L.; Winters, R.; Nilson, J.; Booth, K. Building a Computer Program Grader. *Inf. Syst. Educ. J.* **2005**, *3, 1–16* .
122. Edwards, S.H.; Perez-Quinones, M.A. Web-CAT: Automatically Grading Programming Assignments. In Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education, Madrid, Spain, 30 June–2 July 2008; ACM: New York, NY, USA, 2008; p. 328.
123. Chen, H.M.; Nguyen, B.A.; Yan, Y.X.; Dow, C.R. Analysis of Learning Behavior in an Automated Programming Assessment Environment: A Code Quality Perspective. *IEEE Access* **2020**, *8*, 167341–167354. [CrossRef]

124. Pribela, I.; Pracner, D.; Budimac, Z. Bringing Together Manual and Automated Code Assessment. In Proceedings of the 2015 AIP Conference 1648, Rhodes, Greece, 22–28 September 2014.
125. Edwards, S.H. 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, Norfolk, VA, USA, 3–7 March 2004; ACM: New York, NY, USA, 2004; pp. 26–30.
126. Schrittwieser, S.; Katzenbeisser, S.; Kinder, J.; Merzdovnik, G.; Weippl, E. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis. *ACM Comput. Surv.* **2017**, *49*, 1–37. [CrossRef]
127. Pettit, R.; Prather, J. Automated Assessment Tools: Too Many Cooks, not Enough Collaboration. *J. Comput. Sci. Coll.* **2017**, *32*, 113–121.