A Systematic Literature Review of Automated Feedback Generation for Programming Exercises

HIEKE KEUNING, Open University of the Netherlands and Windesheim University of Applied Sciences
JOHAN JEURING, Utrecht University and Open University of the Netherlands
BASTIAAN HEEREN, Open University of the Netherlands

Formative feedback, aimed at helping students to improve their work, is an important factor in learning. Many tools that offer programming exercises provide automated feedback on student solutions. We have performed a systematic literature review to find out what kind of feedback is provided, which techniques are used to generate the feedback, how adaptable the feedback is, and how these tools are evaluated. We have designed a labelling to classify the tools, and use Narciss' feedback content categories to classify feedback messages. We report on the results of coding a total of 101 tools. We have found that feedback mostly focuses on identifying mistakes and less on fixing problems and taking a next step. Furthermore, teachers cannot easily adapt tools to their own needs. However, the diversity of feedback types has increased over the past decades and new techniques are being applied to generate feedback that is increasingly helpful for students.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Applied computing** → **Computer-assisted instruction**; **Interactive learning environments**;

Additional Key Words and Phrases: Systematic literature review, automated feedback, programming tools, learning programming

ACM Reference format:
Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Trans. Comput. Educ.* 19, 1, Article 3 (September 2018), 43 pages.
https://doi.org/10.1145/3231711

1 INTRODUCTION

Tools that support students in learning programming have been developed since the 1960s [47]. Such tools provide a simplified development environment, use visualisation or animation to give better insight in running a program, guide students toward a correct program by means of hints and feedback messages, or automatically grade the solutions of students [94]. Two important reasons to develop tools that support learning programming are:

This research is supported by the Netherlands Organisation for Scientific Research (NWO), Grant No. 023.005.063.
Authors' addresses: H. Keuning, Open University of the Netherlands and Windesheim University of Applied Sciences; email: hw.keuning@windesheim.nl; J. Jeuring, Utrecht University and Open University of the Netherlands; email: j.t.jeuring@uu.nl; B. Heeren, Open University of the Netherlands; email: bastiaan.heeren@ou.nl.

1946-6226/2018/09-ART3 $15.00
https://doi.org/10.1145/3231711

- learning programming is challenging [125], and students need help to make progress [29];
- programming courses are taken by thousands of students all over the world [20], and helping students individually with their problems requires a huge time investment of teachers [133].

Feedback is an important factor in learning [70, 160]. Boud and Molloy define feedback as "the process whereby learners obtain information about their work to appreciate the similarities and differences between the appropriate standards for any given work, and the qualities of the work itself, to generate improved work" [24]. Thus defined, feedback is formative: It consists of "information communicated to the learner with the intention to modify his or her thinking or behavior for the purpose of improving learning" [160]. Summative feedback in the form of grades or percentages for assessments also provides some information about the work of a learner. However, the information a grade without accompanying feedback gives about similarities and differences between the appropriate standards for any given work, and the qualities of the learner's work, is usually only superficial. In this article, we focus on the formative kind of feedback as defined by Boud and Molloy. Formative feedback comes in many variants, and the kind of formative feedback together with student characteristics greatly influences the effect of feedback [127].

Focussing on the context of computer science education (CSE), Ott et al. [135] provide a roadmap for effective feedback practices for different levels and stages of feedback. The authors see a role for automated feedback at all three levels as defined by Hattie and Timperley [70]: "task level," "process level," and "self-regulation level," discarding feedback at the "self level" because of its limited effect on learning. In their roadmap, automated assessment of exams is placed at the task level, student support through adaptive feedback from automated tools at the process level, and tutoring systems and automated assessment as options for self-assessment of students at the self-regulation level.

Given the role of feedback in learning (programming), we want to find out what kind of feedback is provided by tools that support a student in learning programming. What is the nature of the feedback, how is it generated, can a teacher adapt the feedback, and what can we say about its quality and effect? An important learning objective for learning programming is the ability to develop a program that solves a particular problem. We narrow our scope by only considering tools that offer exercises (also referred to as tasks, assignments, or problems, which we consider synonyms) that let students practice with developing programs.

To answer these questions, we have performed a systematic literature review of automated feedback generation for programming exercises. A systematic literature review (SLR) is "a means of identifying, evaluating and interpreting all available research relevant to a particular research question, or topic area, or phenomenon of interest" [99]. An SLR results in a thorough and fair examination of a particular topic. According to the literature, a research plan should be designed in advance, and the execution of this plan should be documented in detail, allowing insight into the rigorousness of the research.

This article expands on the results of the first iteration of our search for relevant tools, on which we have already reported in a conference paper [96] and a technical report [97]. This first iteration resulted in a set of 69 different tools, described in 102 papers. After slightly adjusting our criteria, the completed search resulted in a final collection of 101 tools described in 146 papers. We searched for papers in related reviews on tools for learning programming and executed multiple steps of 'backward snowballing' by selecting relevant references. We also searched two scientific databases and performed backward snowballing on those results as well.

We have classified the kind of feedback given by the tools we found by means of Narciss' [131] categories of feedback, such as "knowledge about mistakes" and "knowledge about how to

proceed." We have instantiated these feedback categories for programming exercises, and introduce several subcategories of feedback particular to programming. Narciss' categories largely overlap with the categories used to describe the actions of human tutors when they help students learning programming [179]. Next, we determine *how* these tools generate feedback by examining the underlying techniques. Besides looking at feedback categories (the output of a tool) and the technique (what happens inside a tool), we also look at the input. The input of a tool that supports learning programming may take the form of model solutions, test cases, feedback messages, and so on, and determines to a large extent the adaptability of the tool, which is considered important [26, 115]. Finally, we collect information about the effectiveness of the generated feedback. The effectiveness of a tool depends on many factors and tools have been evaluated by a large variety of methods.

This review makes the following contributions:

- An extensive overview of tools that give automated feedback.
- A description of what kind of feedback is used in tools that support a student in learning programming. Although multiple other reviews analyse such tools, none of them specifically looks at the feedback provided by these tools.
- An analysis of the relation between feedback content and its technology, and the adaptability of the tool.

This article expands on our previous conference paper [96] in the following ways:

- We removed 23 tools from our initial set of 69, after adjusting our inclusion criteria based on the initial findings (described in Section 3.2). We completed our search by adding data for 55 new tools.
- We give elaborated examples and descriptions of several of these tools.
- We provide and discuss new tables and graphs summarising our final results. We look at the data more in-depth by identifying trends in time, and combinations of techniques and methods.
- We update, extend and fine-tune the discussion of the results, resulting in a more nuanced conclusion because of the characteristics of more recent tools that were included later.

The article is organised as follows. Section 2 discusses related reviews of tools for learning programming. Section 3 gives our research questions and research method, and Section 4 describes the labelling we developed for coding the tools. The results are described in Sections 5–9, each describing the results for one of the research questions. Section 10 discusses the results and limitations of this review, and Section 11 concludes the article.

2 RELATED WORK

We have found 17 reviews of tools for learning programming, mostly on automated assessment (AA) tools [2, 28, 47, 79, 147, 151, 169] or learning environments for programming [45, 46, 58, 66, 94, 112, 113, 137, 142, 178]. Generating feedback is important for both kinds of tools. Most AA tools only grade student solutions, but some tools also provide elaborated feedback and can be used to support learning [2]. We refer to the technical report on the first phase of our review [97] for a detailed discussion of these related reviews, in which we identified their main research questions, the scope of the selected tools, and the method of data collection.

Most review papers describe the features and characteristics of a selection of tools, identify challenges, and direct future research. Except for the review by Ihantola et al. [79], authors select papers and tools based on unknown criteria. Some mention qualitative factors such as impact (counting citations) or the thoroughness of the evaluation of the tool. Most studies do not strive

for completeness, and the scope of the tools that are described varies greatly. Tools are usually categorised, but there is no agreement on the naming of the different categories. Very few papers discuss technical aspects.

Our review distinguishes itself from the above reviews by focusing on the aspect of generating feedback in programming learning tools, closely examining the different types of feedback messages, and identifying the techniques used to generate them. Furthermore, we employ a more systematic approach than all of the above reviews: we select tools in a systematic way following strict criteria, and we code them using a predetermined labelling.

3 METHOD

Performing an SLR requires an in-depth description of the research method. Section 3.1 discusses our research questions. Section 3.2 describes the criteria that we have set to define the scope of our research. Section 3.3 describes the process for searching relevant papers. Finally, Section 3.4 explains the coding process.

3.1 Research Questions

The following four research questions guide our review on automated feedback generation for programming exercises:

RQ 1 What is the nature of the feedback that is generated?
RQ 2 Which techniques are used to generate the feedback?
RQ 3 How can the tool be adapted by teachers to create exercises and to influence the feedback?
RQ 4 What is known about the quality and effectiveness of the feedback or tool?

3.2 Criteria

There is a growing body of research on tools for learning programming for various audiences with different goals. These goals can be to learn programming for its own sake or to use programming for another goal [94], such as creating a game. Our review focuses on students learning to program for its own sake. We have defined a set of inclusion and exclusion criteria (Table 1) that direct our research and target the characteristics of the papers and the tools described therein.

Although there are many online programming tools giving feedback, we do not include tools for which there are no publications, because we do not know how they are designed. The rationale of our functionality criteria is that the ability to develop a program to solve a particular problem is an important learning objective for learning programming [90]. Because we are interested in improving learning, we focus on formative feedback. We use the domain criteria to focus our review on programming languages used in the industry and/or taught at universities. Many universities teach an existing, textual programming language from the start, or directly after a visual language such as Scratch or Alice. We do not include visualisation tools for programming, because they were surveyed extensively by Sorva et al. [165] in the recent past. However, we do include visualisation tools that also provide textual feedback.

Le and Pinkwart [112] have developed a classification of programming exercises that are supported in learning environments. The type of exercises that a learning tool supports determines, to a large extent, how difficult it is to generate feedback. Le and Pinkwart base their classification on the degree of ill-definedness of a programming problem. Class 1 exercises have a single correct solution, and are often quizlike questions with a single solution or slots in a program that need to be filled in to complete some task. Class 2 exercises can be solved by different implementation variants. Usually a program skeleton or other information that suggests the solution strategy is

Table 1. Criteria for the Inclusion/Exclusion of Papers

	Include	Exclude
General	Scientific publications (journal papers and conference papers) in English. Master's theses, PhD theses, and technical reports only if a journal or conference paper is available on the same topic. The publication describes a tool of which at least a prototype has been constructed.	Posters and papers shorter than four pages.
Functionality	Tools in which students work on programming exercises of class 2 or higher from the classification of Le and Pinkwart [112] (see Section 3.2). Tools provide automated, textual feedback on (partial) solutions, targeted at the student.	Tools that only produce a grade, only show compiler output, or return instructor-written feedback.
Domain	Tools that support a high-level, general purpose, textual programming language, including pseudo-code.	Visual programming tools, e.g., programming with blocks and flowcharts. Tools that only teach a particular aspect of programming, such as recursion or multi-threading.
Technology	—	Tools that are solely based on automated testing and give feedback based on test results.

provided, but variations in the implementation are allowed. Finally, class 3 exercises can be solved by applying alternative solution strategies, which we interpret as allowing different algorithms as well as different steps to arrive at a solution.

We select papers and tools that satisfy all inclusion criteria and none of the exclusion criteria. We have included four PhD theses, one Master's thesis, and three technical reports, whose contributions have also been published in a journal or conference paper, because they contained relevant information. Since no review addressing our research questions has been conducted before, and we aim for a complete overview of the field, we consider all relevant papers up to and including the year 2015.

The criterion to exclude tools solely based on automated testing was added after publishing our preliminary results [96] because of the sheer volume of papers that we found. These papers all describe very similar tools, which would make the review too large. Moreover, we do not think that including these papers would provide an interesting contribution within the scope of this review.

3.3 Search Process

The starting point of our search for papers was the collection of 17 review papers described in Section 2. Two authors of this SLR independently selected relevant references from these reviews. Then two authors independently looked at the full text of the papers in the union of these selections, to exclude papers not meeting the criteria. After discussing the differences, we assembled a final list of papers for this first iteration. Following a "backwards snowballing" approach, one

author searched for relevant references in the papers found in the first iteration. This process was repeated until no more new papers were found. We believe that one author is sufficient for this task, because the scope had already been established.

Next, we searched two databases to identify more papers of interest, and to discover more recent work. We have selected a computer science database (ACM Digital Library) and a general scientific database (Scopus). We used the search string from Listing 1 on title, abstract and key words, slightly adjusted for each database.

```
    ( exercise OR assignment OR task OR ( solv * AND problem ))
AND programming
AND (   ( tutor OR tutoring )
     OR (( learn OR teach ) AND ( tool OR environment ))
     OR (( automat * OR intelligent OR generat *)
              AND ( feedback OR hint ))
    )
```

Listing 1. Database search string.

Although the query could have been adjusted so that it would have returned more papers that match our criteria, this adjustment would also have generated a much larger number of irrelevant results. We believe the final query has a good enough balance between accuracy and breadth, and because we also traced references we had an alternative way to find papers that we would have missed otherwise.

The results of the Scopus search were partly inspected by two authors, who separately selected papers by inspecting the title, abstract, key words, and the name of the journal or conference. We combined the results and discussed all differences. In the event of disagreement, we included the paper for further inspection. Two authors then further refined the acquired list by examining the full text separately and again discussing differences. The second part of the Scopus search results, the ACM search results, and the relevant references from both searches were inspected by one author, only consulting another author if there were doubts.

We had to exclude a small number of papers we could not find after an extensive search and, in some cases, contacting the authors. Some excluded papers point to a potentially interesting tool. We checked if these papers mention a better reference that we could add to our selection.

When we encountered papers we did not trust, we looked further into its contents, author, or the journal that published it. We excluded one paper from the review after all authors agreed that the paper was unreliable and would have a negative influence on the quality of our review (this particular paper seemed to be a copy of existing work).

Often, multiple papers have been written on (versions of) a single tool. We searched for all publications on a tool by looking at references from and to papers already found, and searching for other relevant publications by the authors. We selected the most recent and complete papers about a tool. We prefer journal papers over conference papers, and conference papers over theses or technical reports. All papers from which we collected information appear in our reference list.

Table 2 shows the number of papers found by the searches. Many papers appeared multiple times in our search, both in references and in database searches. The table only counts a tool when it first appeared in the search, which was conducted in the order of the sources in the table.

3.4 Coding

To systematically encode the information in the papers, we developed a labelling (see Section 4) based on the answers to the research questions we expected to get, refined by coding a small set of randomly selected papers. One of the authors coded the complete set of papers. Whenever there

Table 2. Results of Database Search and Snowballing

		Snowballing iterations**				
Source	**Papers***	**1st**	**2nd**	**3rd**	**4th**	**Total**
Review papers	—	46 (76)	15 (17)	6 (9)	2 (2)	69 (104)
Scopus database	1830	25 (35)	5 (5)	—	—	30 (40)
ACM Digital library	798	2 (2)	—	—	—	2 (2)
						101 (146)

Our previous work [96] included the 46 tools from the first iteration of review papers, and 23 additional tools that we excluded after adjusting the criteria for this final review.
*Excluding duplicates and invalid entries.
**Number of tools (number of papers).

were questions about the coding of a paper, another author checked. In total, 24.8% of the codings were (partly) checked by another author. Most of the checks were done in the earlier stages of the review. A third author joined the general discussions about the coding. When necessary, we adjusted the labelling.

4 LABELLING

This Section describes the labels used for our coding.

4.1 Feedback Types (RQ1)

Narciss [131] describes a "content-related classification of feedback components" for computer-based learning environments, in which the categories target different aspects of the instructional context, such as task rules, errors, and procedural knowledge. We use these categories and extend them with representative subcategories identified in the selected papers. Narciss also considers the *function* (cognitive, meta-cognitive, and motivational) and *presentation* (timing, number of tries, adaptability, and modality) of feedback, which are related to the effectiveness of tutoring. We do not include these aspects in our review, because it is often unclear how a tool or technique is used in practice (e.g., formative or summative).

Narciss first identifies three *simple* feedback components:

- Knowledge of performance for a set of tasks (KP): summative feedback on the achieved performance level after doing multiple tasks, such as "15 of 20 correct" and "85% correct."
- Knowledge of result/response (KR): feedback that communicates whether a solution is correct or incorrect. We identify the following meanings of correctness of a programming solution: (1) it passes all tests, (2) it is equal to a model program, (3) it satisfies one or more constraints, (4) a combination of the above.
- Knowledge of the correct results (KCR): a description or indication of a correct solution.

These types of feedback are not intended to "generate improved work," a requirement in the feedback definition by Boud and Molloy. Moreover, Kyrilov and Noelle [104] have investigated the effect of instant binary feedback (messages that either contain "correct" or "incorrect") in automated assessment tools and found harmful effects on student behaviour. They found that students who received this kind of messages plagiarised more often and attempted fewer exercises. Because we focus on formative feedback on a single exercise, we do *not* identify these types in our coding.

The next five types are *elaborated* feedback components. Each type addresses an element of the instructional context. Below, we describe these types and their subtypes in detail.

Knowledge About Task Constraints (KTC). This type focusses on the task itself and is subdivided into two subtypes:

- Hints on task requirements (TR). A task requirement for a programming exercise can be to use a particular language construct or to not use a particular library method.
- Hints on task-processing rules (TPR). These hints provide general information on how to approach the exercise and do not consider the student's current work.

Narciss gives a larger set of examples for this type of feedback, such as "hints on type of task." We do not identify this type, because the range of exercises is limited by our scope. Also, we do not identify "hints on subtasks" as a separate category, because the exercises we consider are relatively small. Instead, we label these hints with KTC-TPR.

Knowledge About Concepts (KC). We Distinguish Two Subtypes:

- Explanations on subject matter (EXP), generated while a student is working on an exercise.
- Examples illustrating concepts (EXA).

Knowledge About Mistakes (KM). KM feedback messages have a type and a level of detail. The level of detail can be *basic*, which can be a numerical value (total number of mistakes, grade, percentage), a location (line number, code fragment), or a short type identifier such as "compiler error"; or *detailed*, which is a description of the mistake, possibly combined with some basic elements. We use five different labels to identify the type of the mistake:

- Test failures (TF). A failed test indicates that a program does not produce the expected output.
- Compiler errors (CE). Compiler errors are syntactic errors (incorrect spelling, missing brackets) or semantic errors (type mismatches, unknown variables) that can be detected by a compiler and are not specific for an exercise.
- Solution errors (SE). Solution errors can be found in programs that do not show the behaviour that a particular exercise requires, and can be runtime errors (the program crashes because of an invalid operation) or logic errors (the program does not do what is required), or the program uses an alternative algorithm that is not accepted.
- Style issues (SI). In various papers, we have found different definitions of programming style issues, ranging from formatting and documentation issues (e.g., untidy formatting, inconsistent naming, lack of comments) to structural issues and issues related to the implementation of a certain algorithm (use of control structures, elegance).
- Performance issues (PI). A student program takes too long to run or uses more resources than required.

Knowledge About How to Proceed (KH). We identify three labels in this type. Each of these types of feedback has a level of detail: a *hint* that may be in the form of a suggestion, a question, or an example; a *solution* that directly shows what needs to be done to correct an error or to execute the next step; or both hints and solutions.

- Bug-related hints for error correction (EC). Sometimes it is difficult to see the difference between KM feedback and EC. We identify feedback as EC if the feedback clearly focuses on what the student should do to correct a mistake.
- Task-processing steps (TPS). This type of hint contains information about the next step a student has to take to come closer to a solution.
- Improvements (IM). This type deals with hints on how to *improve* a solution, such as improving the structure, style or performance of a correct solution. However, if style- or performance-related feedback is presented in the form of an analysis instead of a suggestion

for improvement, we label it as KM. The IM label has been added after we published the results of the first iteration of our search [96].

Knowledge About Meta-cognition (KMC). Meta-cognition deals with a student knowing which strategy to use to solve a problem, if the student is aware of their progress on a task, and if the student knows how well the task was executed. According to Narciss, this type of feedback could contain "explanations on metacognitive strategies" or "metacognitive guiding questions."

4.2 Technique (RQ2)

We distinguish between general techniques for Intelligent Tutoring Systems (ITSs) and techniques specific for the programming domain. Each category has several subcategories.

General ITS Techniques.

- Tools that use model tracing (MT) trace and analyse the process that the student is following solving a problem. Student steps are compared to production rules and buggy rules [128].
- Constraint-based modelling (CBM) only considers the (partial) solution itself, and does not take into account how a student arrived at this (partial) solution. A constraint-based tool checks a student program against predefined solution constraints, such as the presence of a for-loop or the calling of a method with certain parameters, and generates error messages for violated constraints [128].
- Tutors based on data analysis (DA) use large sets of student solutions from the past to generate hints. This type was also added after publishing our first results [96].

Domain-specific Techniques for Programming.

- Dynamic code analysis using automated testing (AT). The most basic form of automated testing is running a program and comparing the output to the expected output. More advanced techniques are unit testing and property-based testing, often implemented using existing test frameworks, such as JUnit.
- Basic static analysis (BSA) analyses a program (source code or bytecode) without running it, and can be used to detect misunderstood concepts, the absence or presence of certain code structures, and to give hints on fixing these mistakes [169].
- Program transformations (PT) transform a program into another program in the same language or a different language. An example is normalisation: transformation into a sublanguage to decrease syntactical complexity. Another example is migration: transformation into another language at the same level of abstraction.
- Intention-based diagnosis (IBD) uses a knowledge base of programming goals, plans or (buggy) rules to match with a student program to find out which strategy the student uses to solve an exercise. IBD has some similarities to CBM and static analysis, and some solutions are borderline cases. Compared to CBM, IBD provides a more complete representation of a solution that captures the chosen algorithm.
- External tools (EX) other than testing tools, such as standard compilers or static code analysers. These tools are not the work of the authors themselves and papers do not usually elaborate on the inner workings of the external tools used. If a tool uses automated testing, for which compilation is a prerequisite, then we do not use this label.

4.3 Adaptability (RQ3)

We identify several input types for tools that enable teachers to create exercises and influence the generated feedback.

- Solution templates (ST) (e.g., skeleton programs and projects) presented to students for didactic or practical purposes as opposed to technical reasons such as easily running the program.
- Model solutions (MS) are correct solutions to a programming exercise.
- Test data (TD), by specifying program output or defining test cases.
- Error data (ED) such as bug libraries, buggy solutions, buggy rules and correction rules. Error data usually specify common mistakes for an exercise.

Another aspect we consider is the adaptability of the feedback generation based on a student model (SM). A student model contains information on the capabilities and level of the student, and may be used to personalise the feedback.

4.4 Quality (RQ4)

As a starting point for collecting data on the quality of tools, we have identified and categorised how tools are evaluated. Tools have been evaluated using a large variety of methods. We use the three main types for the assessment of tools distinguished by Gross and Powers [61].

- Anecdotal (ANC). Anecdotal assessment is based on the experiences and observations of researchers or teachers using the tool. We will not attach this label if another type has been applied as well, because we consider anecdotal assessment to be inferior to the other types.
- Analytical (ANL). Analytical assessment compares the characteristics of a tool to a set of criteria related to usability or a learning theory.
- Empirical assessment. Empirical assessment analyses qualitative data or quantitative data. We distinguish three types of empirical assessment:
  - —Looking at the learning outcome (EMP-LO), such as mistakes, grades and pass rates, after students have used the tool, and observing tool use.
  - —Student and teacher surveys (EMP-SU) and interviews on experiences with the tool.
  - —Technical analysis (EMP-TA) to verify whether a tool can correctly recognise (in)correct solutions and generate appropriate hints. Tool output for a set of student submissions can be compared to an analysis by a human tutor.

5 GENERAL TOOL CHARACTERISTICS

In this section, we discuss the general characteristics of the tools we investigated, such as their type, supported programming language and exercises. Table 3 shows an overview of these characteristics and the papers we consulted for each tool. The complete coding is available as an appendix to this article and as a searchable online table.[1]

In the remainder of this article, we only cite papers on tools in specific cases. We refer to tools by their name in Small caps, or the first author and year of the most recent paper (Author00) on the tool we have used.

History. Figure 1 gives an impression of when the tools appeared in time. Because we do not know exactly in which time frame tools were active, we calculated the rounded median year of the publications related to a tool that we used for our review. Between the 1960s and 1980s a small number of tools appeared. Since the 1990s, we can see an increase in the number of tools, which slowly grows in the 2000s and 2010s.

Tool Types. The tools that fall within our criteria are mostly either Automated Assessment (AA) systems or Intelligent Tutoring Systems (ITSs). AA systems focus on assessing a student's final

[1] www.hkeuning.nl/review.

Table 3. Tool Publications, Supported Language Paradigm, and Exercise Class

Tool	Publications	Language	Class	Tool	Publications	Language	Class
ACT Programming Tutor (APT)	[34, 35, 36]	Multi	2	(Jackson00)	[82]	Imp/OO	3
ADAPT	[53]	Log	3	Java Sensei	[14]	Imp/OO	2
(Ala-mutka04)	[3]	Unknown	3	(Jin12)	[85]	Imp/OO	3
(Allemang91)	[5]	Imp/OO	3	(Jin14)	[86]	Imp/OO	2
AnalyseC	[194]	Imp/OO	3	JITS	[171, 172]	Imp/OO	2
APROPOS2	[114]	Log	3	(Keuning14)	[95]	Imp/OO	3
Ask-Elle	[54, 55, 84]	Fun	3	(Kim98)	[98]	Imp/OO	3
ASSYST	[81, 83]	Imp/OO	3	Koh	[100]	Imp/OO	2
At(x)	[15, 16]	Multi	3	LAURA	[1]	Imp/OO	2
AutoGrader	[72]	Imp/OO	3	(Lazar14)	[106]	Log	3
AutoLEP	[188]	Imp/OO	3	LISP tutor	[7, 37]	Fun	2
AutoStyle	[129]	Imp/OO	3	Ludwig	[158]	Imp/OO	3
AutoTeach	[8, 9]	Imp/OO	3	(Mandal07)	[119]	Imp/OO	3
BIP	[11–13]	Imp/OO	3	Marmoset	[166]	Multi	3
Bridge	[23]	Imp/OO	2	MENO-II	[163]	Imp/OO	3
C-tutor	[164]	Imp/OO	3	(Naur64)	[132]	Imp/OO	3
Camus	[182]	Imp/OO	3	Online Judge	[31]	Imp/OO	3
Ceilidh	[17–19]	Multi	3	PASS	[173]	Imp/OO	2
(Chang00)	[30]	Imp/OO	2	PATTIE	[40]	Imp/OO	3
Checkpoint	[50]	Multi	3	Pex4Fun	[174]	Multi	3
CHIRON	[155]	Imp/OO	3	PHP ITS	[192, 193]	Imp/OO	3
COALA	[91, 92]	Imp/OO	2	PRAM	[74, 120]	Log	3
Code Hunt	[138, 139]	Imp/OO	3	ProgramCritic	[154]	Imp/OO	3
CourseMarker/CourseMaster	[52, 73, 75]	Multi	3	ProgTest	[42–44]	Imp/OO	3
CSTutor	[67, 68]	Imp/OO	3	ProPAT_deBUG	[41]	Imp/OO	3
Ctutor	[103]	Imp/OO	2	ProPL	[105]	Imp/OO	2
(Dadic11)	[38, 39]	Imp/OO	3	PROUST	[87, 155]	Imp/OO	3
datlab	[116, 117]	Imp/OO	3	(Rosenthal02)	[152]	Imp/OO	3
DISCOVER	[148]	Imp/OO	2	(Ruth76)	[153]	Imp/OO	3
EASy	[118]	Imp/OO	3	(Sant09)	[157]	Imp/OO	3
ELM-PE/ELM-ART (II)	[27, 189–191]	Fun	3	SCENT	[123, 124]	Fun	3
ELP	[175, 176]	Imp/OO	2	Scheme-robo	[156]	Fun	3
(Fischer06)	[51]	Imp/OO	3	(Shimic12)	[159]	Imp/OO	2
FIT Java Tutor	[62, 63, 64]	Imp/OO	3	(Singh13)	[161]	Imp/OO	3
FLIP	[93]	Imp/OO	3	SIPLeS-II	[195]	Imp/OO	3
GAME (2, 2+)	[21, 22, 121, 122]	Multi	3	Smalltalker	[32]	Imp/OO	2
(Ghosh02)	[56]	Imp/OO	3	SOP	[162]	Imp/OO	3
Grace	[126, 146]	Imp/OO	2	(Striewe11)	[168]	Imp/OO	3
(Gulwani14)	[65]	Imp/OO	3	submit	[183]	Imp/OO	3
HabiPro	[185, 186]	Imp/OO	2	Submit!	[143]	Multi	3
(Hasan88)	[69]	Imp/OO	3	Talus	[130]	Fun	3
(He94)	[71]	Imp/OO	2	Test My Code	[136, 184]	Imp/OO	3
(Hong04)	[77]	Log	3	Testovid	[144]	Multi	3

(Continued)

Table 3. Continued

Tool	Publications	Language	Class	Tool	Publications	Language	Class
INCOM	[108, 110, 111]	Log	3	Ugo	[80]	Log	3
InSTEP	[134]	Imp/OO	2	VC Prolog Tutor	[141]	Log	3
INTELLITUTOR (II)	[177]	Imp/OO	2	Virtual Programming Lab	[150]	Multi	3
IPTS	[196]	Imp/OO	2	(Vujosevic-Janicic13)	[187]	Imp/OO	3
ITAP	[149]	Imp/OO	3	Web-CAT	[48, 49]	Multi	3
ITEM/IP	[25]	Imp/OO	3	WebToTeach	[10]	Imp/OO	3
J-LATTE	[76]	Imp/OO	2	WebWork-JAG	[59, 60]	Imp/OO	3
JACK	[57, 101, 102, 167]	Imp/OO	3				

Class 2 tools support exercises that can be solved by a predetermined strategy, allowing small variations. Exercises in Class 3 tools can be solved by multiple strategies (see Section 3.2).

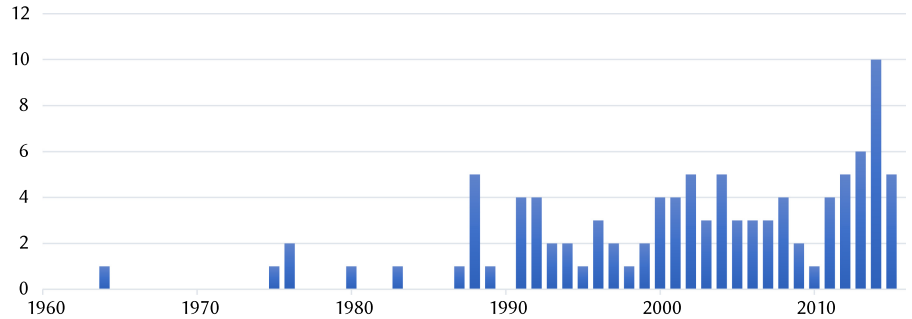


Fig. 1. Number of tools by median publication year.

solution to an exercise with a grade or a feedback report, to alleviate instructors from manually assessing a large number of students. ITSs help students to arrive at the solution by offering help at each step [181]. Other tools we found are programming environments for novices with a feedback component. A newer type of such a tool is the educational programming game, or serious game, in which learning programming is more implicit and considered a side effect of playing a fun game. Examples are Pex4Fun and its successor Code Hunt that challenge students to iteratively discover the specification of a hidden program. Code Hunt has an even stronger gaming vibe to it, created by the "worlds" and "levels" in which the player solves programming problems.

Some papers more generally describe a technique that can be used for generating feedback, which could be applied in an assessment or learning tool. We came across many papers on debugging and program understanding techniques, of which we only included the ones that were clearly used in an educational context.

Programming Language. Tools offer exercises for a specific programming language, a set of programming languages within a particular paradigm, or multiple languages within multiple paradigms. Table 4 shows the distribution of the different paradigms. The majority of the 101 tools supports programming in imperative languages, including object-oriented languages. Tools developed in the 21st Century often support imperative languages such as Java, C, and C++, whereas older tools provide exercises in ALGOL (Naur64), FORTRAN (LAURA), and Ada (ASSYST). Some recent tools focus on (web) scripting languages such as PHP, JavaScript, and Python. Tools for functional programming languages support Lisp (the LISP tutor), Scheme (Scheme-robo), or Haskell (Ask-Elle). All tools for logic programming offer exercises in Prolog. The remaining tools support multiple languages of different types and paradigms and are often test-based AA systems.

Table 4. Supported Language Paradigm of Tools (n = 101)

Paradigm	%
Imperative/object-oriented	73.3
Multiple paradigms	11.9
Logic	7.9
Functional	5.9
Unknown	1.0

Programming exercises often require a student to write a few lines of code or a single function, meaning that many tools only support a subset of the features of a programming language. For instance, a tool that requires programming in Java, an object-oriented language, may not support feedback generation on class declarations.

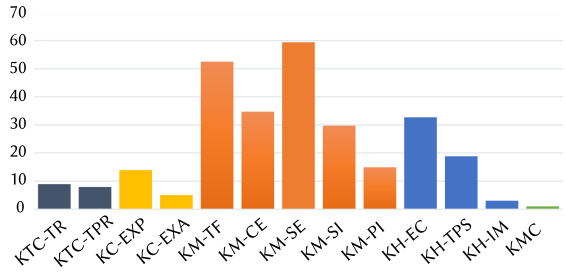
Exercise Type. We have recorded the highest exercise class a tool supports and found that 23.8% of the tools support exercises of class 2 and 76.2% exercises of class 3. However, exercises that do not require a student to write code him or herself do not easily fit in this classification. For example, in PATTIE and ProPL the student engages in a conversation with an automated tutor to solve a programming problem. If the student makes a good suggestion, then the tutor expands the solution-in-progress. CHIRON, the successor to the intelligent debugger PROUST also incorporates an interactive question/answer session with the student. Through natural language parsing CHIRON responds to questions on topics such as terminology, goal/plan implementation and data flow. We have included these systems, because this approach closely mimics the behaviour of a human tutor.

Another uncommon type of exercise is offered by the programming game Code Hunt that asks the student to write a program for which he or she does not know the specification yet. Instead, the student has to discover the specification by inspecting the results of a set of test cases and modifying their code to satisfy these tests.

6 FEEDBACK TYPES (RQ1)

This section describes the results of the first research question: "What is the nature of the feedback that is generated?" Figure 2 shows for each feedback type the percentage of tools that offer it, including the distinction between class 2 and class 3 tools. The percentages do not add up to 100%, because one tool can provide more than one feedback type. We have found KTC (knowledge about task constraints) feedback and KC (knowledge about concepts) feedback in only a few tools, with 14.9% and 16.8%, respectively. KM (knowledge about mistakes) is by far the largest type: with 96.0% it is found in almost all tools. The subtype of KM we found most in the first part of our review, was TF (test failures). After omitting purely test-based tools in this final review, SE (solution errors) is the largest category with 59.4%. We have found KH feedback (knowledge about how to proceed) in 44.6% of the tools, of which EC feedback (error correction) is the largest subcategory with 32.7%.

Class 2 tools provide more solution error feedback (83.3% versus 51.9%), and more KH feedback (knowledge about how to proceed) with around twice as much for each subtype. Class 3 tools provide more test failure feedback (59.7% versus 29.2%) and more style issue feedback (36.4% versus 8.3%) and performance issue feedback (19.5% versus 0%). KTC (knowledge about task constraints) feedback is also seen in fewer class 2 tools. We also calculated that in 11.9% of the tools one type of feedback is generated, but 48.5% of the tools only generate one of the five main types (these



	All (n=101)		Class 2 (n=24)		Class 3 (n=77)	
Feedback type	Count	%	Count	%	Count	%
KTC Knowledge about task constraints	*15*	*14.9*	*2*	*8.3*	*13*	*16.9*
TR Hints on task requirements	9	8.9	1	4.2	8	10.4
TPR Hints on task-processing rules	8	7.9	1	4.2	7	9.1
KC Knowledge about concepts	*17*	*16.8*	*5*	*20.8*	*12*	*15.6*
EXP Explanations on subject matter	14	13.9	3	12.5	11	14.3
EXA Examples illustrating concepts	5	5.0	2	8.3	3	3.9
KM Knowledge about mistakes	*97*	*96.0*	*24*	*100.0*	*73*	*94.8*
TF Test failures	53	52.5	7	29.2	46	59.7
CE Compiler errors	35	34.7	8	33.3	27	35.1
SE Solution errors	60	59.4	20	83.3	40	51.9
SI Style issues	30	29.7	2	8.3	28	36.4
PI Performance issues	15	14.9	0	0.0	15	19.5
KH Knowledge about how to proceed	*45*	*44.6*	*16*	*66.7*	*29*	*37.7*
EC Bug-related hints for error correction	33	32.7	13	54.2	20	26.0
TPS Task-processing steps	19	18.8	7	29.2	12	15.6
IM Improvement hints	3	3.0	1	4.2	2	2.6
KMC Knowledge about meta-cognition	*1*	*1.0*	*1*	*4.2*	*0*	*0.0*

Fig. 2. Number of tools (count) and percentage of tools that offer a feedback type (by subtype and combined by main type), for all tools, and subdivided by exercise class. A tool can offer more than one feedback type.

percentages are not in the table). In the following subsections, we expand on the different feedback types and provide examples of tools and the feedback messages they provide.

6.1 Knowledge about Task Constraints (KTC)

The first things a student should know when attempting an exercise are the requirements of the task and possibly some information on how to approach it.

6.1.1 Hints on Task Requirements (TR). An example of this subtype can be found in the INCOM system. When a student makes a mistake with implementing the method header, feedback is given by "highlighting keywords in the task statement and advising the student to fully make use of the available information" [109].

Another example can be found in the BASIC Instructional Program (BIP), a system from the seventies. Some exercises in BIP require the use of a specific language construct. If this construct is missing from the student solution, then the student will see the following message [11]:

```
Wait. Something is missing.
For this task, your program should also include the following basic
statement(s): FOR
```

Automated assessment tools have to check for task requirements as well. For example, if an exercise requires implementing a method that is also available in the standard library of the language, the assessment tool will have to check if this library method was not used. The AA tool from Fischer06 provides the following feedback when a student uses a prohibited method [51]:

```
signature and hierarchy: failed
Invocation test checks whether prohibited classes or methods are used; call of
method reverse from the prohibited class java.lang.StringBuffer
```

6.1.2 Hints on Task-processing Rules (TPR). For this category, we consider the built-in feedback related to the tutoring strategy of the tool, as opposed to the dynamically generated feedback based on a (partial) student solution that we categorise under KM and KH. Tools that provide this kind of feedback may have a built-in stepwise approach to solving a programming problem. The ACT programming tutor shows a "skill meter" that indicates the probability that a student knows a "rule" (a rule corresponds to a step). The automated tutor in ADAPT gives some general information on how to solve a particular exercise [53]:

```
There are 2 major components to this template:
• base case
• recursive step
For the base case, the basic idea is to stop processing when the list becomes
empty and return 0 for the sum. For the recursive step, the basic idea is to
[...]
```

6.2 Knowledge About Concepts (KC)

This feedback type deals with the subject matter of the exercise.

6.2.1 Explanations on Subject Matter (EXP). This subtype can be found in the Ask-Elle tutor that refers to relevant internet sources when a student encounters certain language constructs.

6.2.2 Examples Illustrating Concepts (EXA). The LISP tutor uses this type of feedback in its tutoring dialogue. After a student has made a mistake, the tutor might respond with [7]:

```
That is a reasonable way to think of doing factorials, but it is not a
plan for a recursive function. Since you seem to be having trouble with the
recursive cases, let us work through some examples and figure out the
conditions and actions for each of these cases.
```

The ELM-PE/ART tutors support "example-based programming" and provide a student with an example program that the student solved in the past, specifically selected to help the student solve a new problem.

The FIT Java Tutor uses machine learning techniques to generate example-based feedback, as shown in Figure 3. The program on the right is an example program (KC-EXA) for the student who programmed the erroneous program on the left. Differences with respect to the example program may be highlighted in the student program (KM-SE), and a feedback message tells the student that the program on the right is one step further (KH-TPS). The student can compare the two programs and identify mistakes and next steps. The data set consists of both student solutions and sample solutions by experts, from which a representative solution is chosen to compare with the student

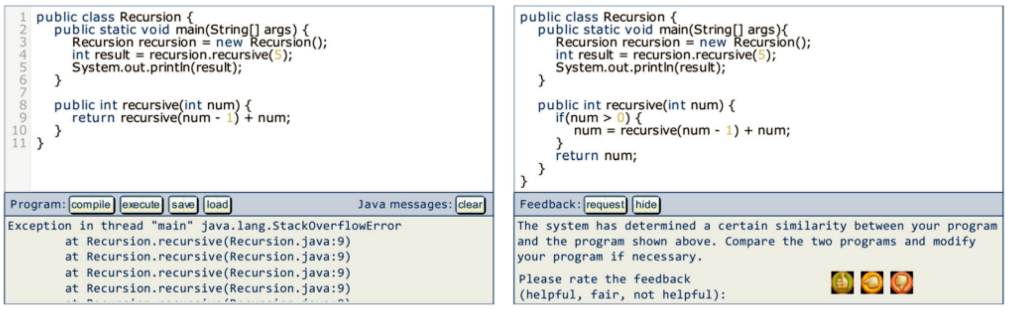


Fig. 3. Feedback from the FIT Java Tutor (image from Reference [62]).

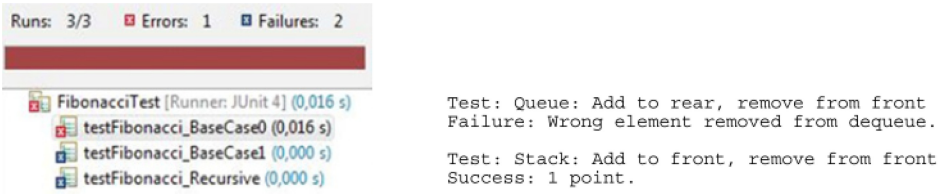


Fig. 4. Feedback showing test cases in COALA on the left (image fragment from Reference [91]) and feedback on test execution in Testovid on the right (image fragment from Reference [144]).

solution. In situations in which no representative solution can be selected, the student solution is compared to a similar program from the data set. However, it is possible that the similar program is not correct, in which case the feedback addresses the possible incorrectness and asks the student to identify mistakes in the similar program.

6.3 Knowledge About Mistakes (KM)

This most common feedback type deals with reporting mistakes to students.

6.3.1 Test Failures (TF). A large number of tools give feedback based on the success and failure of executing tests on student programs. Online Judge is an automatic judge for programming contests that provides basic feedback on test failures. The system returns a short string such as "`[..x]`" as feedback, indicating that tests cases 1 and 2 are successful (indicated by a dot) and test case 3 is not successful (indicated by an "x").

Figure 4 shows two examples of more detailed feedback on test execution. COALA shows the output of JUnit test cases integrated in a customised Eclipse environment. Testovid provides more informative feedback on the execution of test cases. We found this type of feedback, which resembles the output of professional testing tools, in many AA tools.

ProgTest requires a student to upload his or her own tests together with a solution. The tool shows the results of testing the student's code with the student's tests and the instructor's tests but also the results of testing the model program of the instructor with the tests of the student. Additionally, the tool presents the results of a code coverage analysis. The authors of ProgTest have put effort into improving the understandability of the test coverage output. Based on this output, they calculate six metrics, such as "testing completeness," "program correctness," and "tests adequacy," the results of which indicate how well the student performed on different aspects of testing. Instructor-written hints on failed tests can be shown as well.

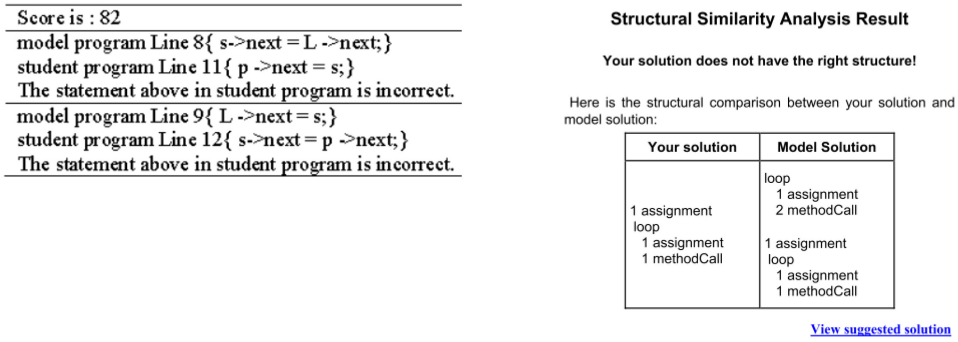


Fig. 5. Solution error feedback in AnalyseC on the left (image from Reference [194]) and ELP on the right (image fragment from Reference [176]).

6.3.2 Compiler Errors (CE). Feedback on compiler errors might be the output of a compiler that is passed on to the student, enabling the student to do exercises without directly using a compiler him or herself. The main reasons for working without a compiler are not having access to the necessary tools and avoiding the difficulty of the compilation process, as experienced by a novice programmer. Test-based AA systems often provide compiler output as feedback, because successful compilation is a prerequisite for executing tests.

Some tools have replaced a standard compiler or interpreter by a more student-friendly alternative. An example is the interpreter used in BIP, which generates extensive error messages in understandable language. Below, we give an example of such a message [11]:

```
*20 PRINT "THE INDEX IS; I
          ↑
SYNTAX ERROR: UNMATCHED QUOTE MARKS -- FOUND NEAR '"THE INDEX IS' LINE NOT
ACCEPTED (TYPE ? FOR HELP)
```

More feedback can be given if the student asks for more help [11]:

```
*?
'"THE INDEX IS' HAS AN ODD NUMBER OF QUOTE MARKS.
REMEMBER THAT ALL STRINGS MUST HAVE A QUOTE AT THE BEGINNING AND END.
```

6.3.3 Solution Errors (SE). We have seen many instances of feedback on solution errors. AutoLEP describes the results of matching the student program with several model programs, comparing aspects such as size, structure, and statements. The tool by Singh13 also produces this type [161]:

```
The program requires 1 change:
- In the function computeDeriv, add the base case at the top to return [0] for
len(poly)=1
```

This feedback message provides a numerical value (the number of required changes) and the location of a solution error. The message does not give a description of what the student has done wrong. The suggestion on how to correct the mistake is labelled with KH, which is described next.

AnalyseC provides detailed feedback on solution errors by identifying incorrect statements and showing the correct statement from a model solution, shown in Figure 5. ELP (also in Figure 5) matches a model solution with the student solution at a slightly higher level by comparing the structure (the language constructs used, such as loops and assignments) of a student solution to a model solution.

```
APPLYING STYLE METRICS...

 12.7  characters per line     :   9.0
       (max   9)
  4.0  % comment lines         :   0.0
       (max  12)
 20.9  % indentation           :   9.7
       (max  12)
 38.5  % blank lines           :   0.0
       (max  11)
  1.6    spaces per line       :   1.5
       (max   8)
```

Fig. 6. Fragment of style feedback with a score for each issue in Jackson00 (image from Reference [82]).

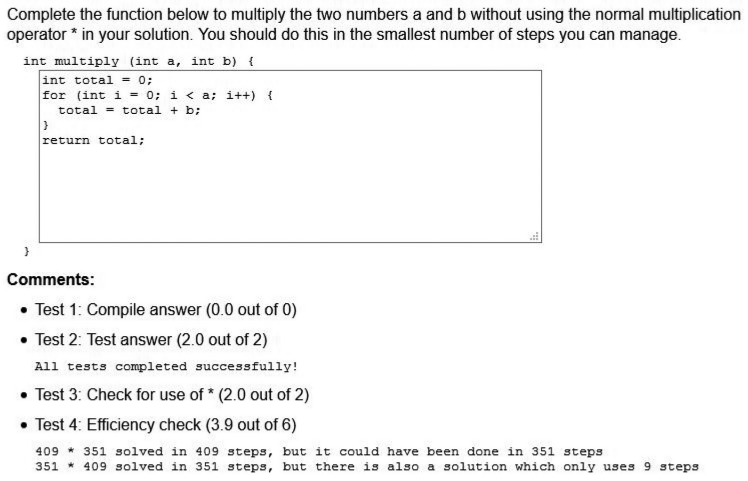


Fig. 7. Feedback on performance in Checkpoint (image fragment from Reference [50]).

Code Hunt takes test-based feedback a step further by combining feedback that shows the results of test cases with a hint that points to a line number on which the code needs to be changed (KM feedback). In addition, the student receives information on new features that are useful to solve the problem, and warns the student against features that might not be such a good idea (KH feedback). Code Hunt uses an approach based on student data: the line number and feature recommendation hints are derived from a large set of previous solutions and unsuccessful attempts.

6.3.4 Style Issues (SI). Many teachers consider learning a good programming style important for novice programmers. As an example of feedback on style issues, Figure 6 shows the feedback generated by the tool of Jackson00 aimed at formatting and commenting.

6.3.5 Performance Issues (PI). Checkpoint, a recent AA system, also provides feedback on performance issues, as can be seen in "Test 4" in Figure 7. Naur64, one of the earliest systems, checks one particular exercise that lets a student write an algorithm for finding the root of a given function. The system gives performance feedback for each test case, such as "No convergence after 100 calls" [132].

6.4 Knowledge About How to Proceed (KH)

Knowing what is wrong with a solution is meaningful information, but to learn how to proceed, students need feedback on fixing their mistakes and taking a next step.

6.4.1 Bug-Related Hints for Error Correction (EC). JITS gives feedback on fixing typing errors, such as "Would you like to replace smu with sum?" [172]. PROUST generates an elaborated error report containing hints on how to correct errors. The following fragment provides such hints [87]:

```
The maximum and the average are undefined if there is no valid input. But lines 34
and 33 output them anyway. You should always check whether your code will work
when there is no input! This is a common cause of bugs.
You need a test to check that at least one valid data point has been input before
line 30 is executed. The average will bomb when there is no input.
```

The examples from ELP and ANALYSEC (Figure 5) in the KM-SE category also contain the correct code of the solution; therefore, we label these tools with KH-EC as well.

CSTUTOR first gives KM-SE feedback in the form of questions that get more specific after each request for help, such as "…Why do you think that the value in the variable "fahrenheit" does not have this value?" [68]. The authors state that asking questions enables the students to reflect on the problem themselves for some time, as opposed to suggesting a solution instantly. If the student continues to ask for help, then the system shows the solution code for the particular problem (KH-EC).

6.4.2 Hints on Task-processing Steps (TPS). Hints on task-processing steps can help a student to solve a programming problem step by step. The Prolog tutor HONG04 provides a guided programming phase. If a student asks for help in this phase, then the tutor will respond with a hint on how to proceed and generates a template for the student to fill in [77]:

```
You can use a programming technique that processes a list until it is empty by
splitting it into the head and the tail, making a recursive call with the tail.

reverse([ ], ⟨arguments.)⟩
reverse([H | T], ⟨arguments⟩) :-
    ⟨pre-predicate⟩,
    reverse(T,⟨arguments⟩),
    ⟨post-predicate⟩.
```

Another example can be found in the ASK-ELLE tutor for functional programming. The tool provides a student with multiple strategies to tackle a programming problem [55]:

```
You can proceed in several ways:
- Implement range using the unfoldr function.
- Use the enumeration function from the prelude.
- Use the prelude functions take and iterate.
```

AUTOTEACH is an incremental hint system that gradually reveals larger parts of a model solution to the student, as shown in Figure 8. The hints are generated by taking a model solution as input and producing output files for multiple hint levels. An output file shows parts of the solution code and replaces hidden code by matching hint messages. However, these hints are pre-processed, meaning that they do not take into account the student's current work. Unit testing is used as a back-up mechanism to check alternative solutions.

The JIN14 programming tutor incorporates two aspects of programming: planning and coding. The "guided-planning" component leads the student step-by-step through several predefined stages such as "variable analysis," "computation," and "output," giving hints along the way. The "assisted-coding" component helps the student with hints on writing code for the different stages. The hint-generation is based on the technique from JIN12 that uses model programs as input.

```
-- Hint level 2.
  sum (a_numbers: ARRAY [INTEGER]): INTEGER
      -- Sum of all elements in 'a_numbers'.
    do
      --# [1] HINT: First, start by initializing Result to zero.
      --#-  This is not necessary, but makes the idea clearer.
      Result := 0

      --# [2] HINT: Maybe you need a loop.

      -- Your code here!

    end
```

Fig. 8. Incremental code revealing feedback in AutoTeach (image from Reference [8]).

6.4.3 Program Improvements (IM). Only a few tools give feedback aimed at program improvements. An example message from Gulwani14 to improve the performance of a program is [65]

`Instead of sorting input strings, compare the number of character occurrences in each string.`

AutoStyle is a tool that gives feedback on programs to make them more concise and readable. It uses historical student data to find a path from a problematic, but functionally correct, solution to a stylistic better solution. As an example, AutoStyle can detect if the functionality of a library method is hand-coded by the student and may suggest using that library method instead.

6.5 Knowledge About Meta-Cognition (KMC)

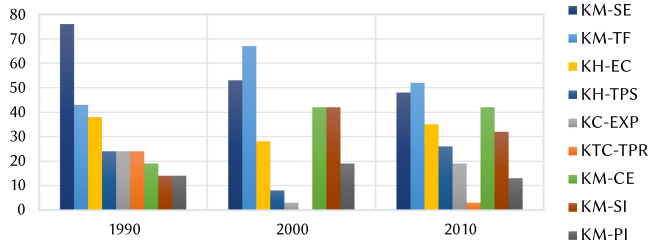
We have only found one example of KMC. HabiPro provides a "simulated student" that responds to a solution by checking if a student really knows why an answer is correct.

6.6 Trends

Figure 9 shows the changes of using a particular type of feedback in a tool over the last three decades. A tool is linked to the decade of its median year (as in Figure 1). We have omitted the decades before the 1990s because of the low number of tools and do not show the types of feedback that never occur in at least 15% of the tools (KTC-TR, KC-EXA, KH-IM, and KMC). The figure shows that in the 1990s 76.2% of the tools provided solution error (KM-SE) feedback, which decreases in the 2000s and 2010s to 52.8% and 48.4%. Test failure feedback (KM-TF) increases in the 2000s from 42.9% to 66.7%, declining to 51.6% in the 2010s. It should be noted, however, that we exclude purely test-based tools, so the actual percentage would be much higher. Multiple types show an increase from the 2000s (KM-CE, KM-SI, KTC-TR), which makes the diversity of feedback types greater in the 21st Century.

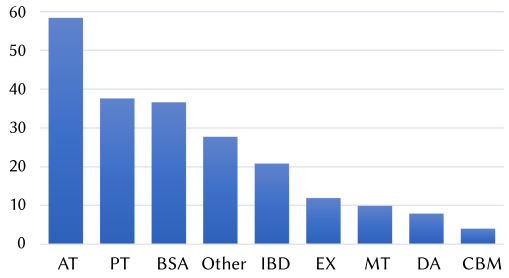
7 TECHNIQUE (RQ2)

This section describes the results of the second research question: "Which techniques are used to generate the feedback?" Figure 10 shows for each technique the percentage of tools that use it. Even after omitting purely test-based tools, automated testing is still the technique used the most (58.4%) in tools that generate feedback. After that, 37.6% use program transformations and 36.6% of all tools use static analysis. Intention-based diagnosis is used in 20.8% and 11.9% of the tools use an external tool. We have found that 9.9% of the tools use model tracing, 7.9% use data analysis and, only 4.0% use constraint-based modelling. In 27.7% of the tools other techniques were found. Figure 10 also shows the differences between tools of class 2 and class 3. Class 3 tools use more automated testing, external tools, intention-based diagnosis and data analysis than class 2



		1990	2000	2010
Feedback type		%	%	%
KM-SE	Solution errors	76.2	52.8	48.4
KM-TF	Test failures	42.9	66.7	51.6
KH-EC	Bug-related hints for error correction	38.1	27.8	35.5
KH-TPS	Task-processing steps	23.8	8.3	25.8
KC-EXP	Explanations on subject matter	23.8	2.8	19.4
KTC-TPR	Hints on task-processing rules	23.8	0.0	3.2
KM-CE	Compiler errors	19.0	41.7	41.9
KM-SI	Style issues	14.3	41.7	32.3
KM-PI	Performance issues	14.3	19.4	12.9

Fig. 9. Percentage of tools with feedback type in the 1990s (n = 21), 2000s (n = 36) , and 2010s (n = 31), omitting types that never occur in at least 15% of the tools. The legend and table have the same order as the bars in the chart.



		All (n=101)		Class 2 (n=24)		Class 3 (n=77)	
Technique		Count	%	Count	%	Count	%
AT	Automated testing	59	58.4	7	29.2	52	67.5
PT	Program transformations	38	37.6	8	33.3	30	39.0
BSA	Basic static analysis	37	36.6	7	29.2	30	39.0
Other		28	27.7	6	25.0	22	28.6
IBD	Intension-based diagnosis	21	20.8	3	12.5	18	23.4
EX	External tools	12	11.9	0	0.0	12	15.6
MT	Model tracing	10	9.9	6	25.0	4	5.2
DA	Data analysis	8	7.9	1	4.2	7	9.1
CBM	Constraint-based modelling	4	4.0	2	8.3	2	2.6

Fig. 10. Number of tools (count) and percentage of tools that employ a technique, for all tools, and subdivided by exercise class.

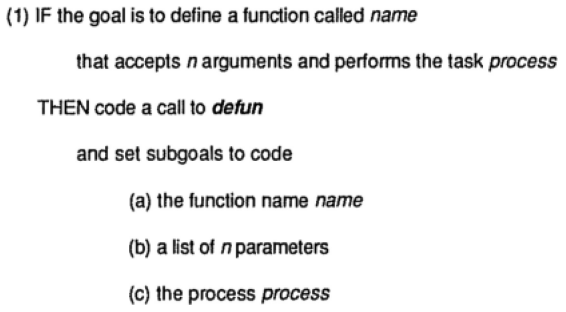


Fig. 11. A production rule from the LISP TUTOR (image from Reference [37]).

tools. Class 2 tools more often use model tracing and constraint-based modelling. In the following subsections, we expand on the different techniques and explain how they are used.

7.1 General ITS Techniques

7.1.1 Model Tracing (MT). The LISP TUTOR is a classic example of a model tracing system. An example of a production rule (rephrased in English) used in this tutor is shown in Figure 11. Whereas classic tools use a production system for model tracing, some tools use a slightly different approach. As an example, ASK-ELLE and KEUNING14 generate *programming strategies* derived from model solutions that are used to check where the student is in his or her programming process and give hints on what to do next. We found one instance of example-tracing used in the JAVA SENSEI tool that offers class 2 exercises. Example-tracing tutors follow the steps that a student takes based on "generalised examples of problem-solving behaviour" [4], making it easier to create ITSs. Although example-tracing is a different tutoring paradigm, we categorise it under the MT label.

7.1.2 Constraint-Based Modelling (CBM). INCOM uses Constraint-based modelling. The authors of INCOM argue that CBM has its limitations in the domain of programming because of the large solution space for programming problems [110]. They have designed a "weighted constraint-based model," consisting of a semantic table, a set of constraints, constraint weights (to indicate the importance of a particular constraint), and transformation rules. The authors show that this model can recognise student intentions in a much larger number of solutions compared to the standard CBM approach.

J-LATTE, the Java Language Acquisition Tile Tutoring Environment, is a constraint-based ITS for learning Java. The tutor presents students with simple programming exercises that can be solved in two modes. In Concept mode, a student selects predefined programming artefacts defined at a higher level (e.g., declarations, return statements, and loops) from the user interface and combines them to create the structure of the solution. After selecting a concept, the accompanying code can be entered in the Coding mode. Using constraints does not force the student to follow a predetermined path. J-LATTE uses constraints to represent domain knowledge describing the syntactic, semantic and style-related features of the solution. An example of a semantic constraint is:

```
(sum-of-function-over-a-range :range (:from (method-arg :name "startNum")
:to (method-arg :name "endNum")) :function square)
```

This specification states that the "sum-of-function-over-a-range" pattern should be used (which could be any kind of loop) with a specific lower and upper limit of the range and a call to a square function. Feedback can be requested by the student at any time during the exercise. The system

presents error messages related to constraints that are violated, such as "You should be initialising the loop-variable to the beginning of the range you are looping over" [76].

7.1.3 Data Analysis (DA). Using large sets of historical student data to generate hints is a recent development that has already produced some promising results. ITAP (Intelligent Teaching Assistant for Programming) is a data-driven tutor for programming in Python. It creates a solution space graph with (intermediate) program states as nodes, in which directed edges represent next steps. ITAP matches a student solution with a state in the graph, constructs a path to a correct solution and generates hints based on the steps of the path.

The Jin12 tool also uses a set of student solutions and creates Markov Decision Processes to generate feedback to correct or finish an incorrect or incomplete program. Lazar14 inspects traces of students solving Prolog programming problems to collect common line edits. This information is then used to find a sequence of line edits that transform an incorrect program into a correct one, preferring the shorter sequences.

7.2 Domain-Specific Techniques for Programming

7.2.1 Dynamic Code Analysis Using Automated Testing (AT). AT is often implemented by running a program and comparing its output to the expected output. Some tools integrate professional testing tools, such as JUnit[2] for unit testing and QuickCheck [33] for property-based testing. Testing is frequently used as a "last resort" if the tool cannot recognise in any other way what the student is doing. Striewe11's technique offers a solution to the problem that students may have difficulty identifying the cause of a failed test case. This is particularly relevant for blackbox testing, which only shows a difference between expected and actual outcome. Striewe11's tool generates run-time traces for the execution of test cases using debugging technology. Students can inspect the traces to find out where unexpected behaviour occurred. For specific issues the tool automatically analyses the trace to provide even more help. For example, assertion checking is used to point to the location of an error, and automated comparison of the student trace to the trace from a model program can be used to detect abnormal behaviour.

Test My Code (TMC) counts bytecode instructions to assess the performance of algorithms. The authors illustrate this with an exercise that requires the student to write an implementation for the Fibonacci sequence that runs in linear time. The resulting calculations can be shown to the student. Counting bytecode instructions as an alternative to simply measuring running time prevents unreliable measurements caused by busy assessment servers.

Test cases may be predefined by the instructor, generated randomly, or have to be supplied by the student him or herself. We have also noticed the use of *reflection* in multiple tools, a technique that can be used to dynamically inspect and execute code. AutoGrader is a lightweight framework that uses Java reflection to execute tests for grading and creating feedback reports.

Feedback on performance issues can be done by profiling, a dynamic program analysis method. PRAM uses profiling to measure several aspects related to complexity in Prolog programs, such as the average number of calls and the percentage of backtracking. The system compares the results to the results of the profiling of a model program and presents the issues found together with a mark to the student.

Finally, code coverage tools can be used to identify unnecessary statements, as seen in ProgTest.

7.2.2 Basic Static Analysis (BSA). Some tools use static analysis for calculating metrics, such as cyclomatic complexity or the number of comments. The GAME-2 tool performs static

[2] www.junit.org.

analysis by examining comments in a solution. The analysis identifies code that is commented out as "artificial" comments, and identifies "meaningful" comments by looking at the ratio of nouns and conjunctions compared to the total word count. INSTEP looks for common errors in code, such as using "=" instead of "==" in a loop condition, or common mistakes in loop counters, and provides appropriate feedback accordingly.

7.2.3 Program Transformations (PT). Transformations are often used together with static code analysis to match a student program with a model program. The normalisation technique used in SIPLES-II is a notable contribution. The authors have identified 13 "semantics-preserving variations" (SPVs) found in code. Some of these SPVs are handled using transformations that change the computational behaviours (operational semantics) of a program while preserving the computational results (computational semantics). As an example, the "different control structures" SPV is handled by transformations that standardise control structures, and the "different redundant statements" SPV by dead code removal. As a result, a larger number of student programs can be recognised.

Migration is applied in INTELLITUTOR, which uses the abstract language AL for internal representation. Pascal and C programs are translated into AL to eliminate language-specific details. After that, the system performs some normalisations on the AL-code.

Synthesis, transformation to a lower level such as bytecode, is another program transformation technique. We have not found this technique in tools other than compilers, and the external tool FindBugs.[3] FindBugs translates Java code into bytecode and performs static analysis on this bytecode to identify potential bugs.

7.2.4 Intention-based Diagnosis (IBD). The term intention-based diagnosis was introduced by Johnson and Soloway [88] for their tutor PROUST. PROUST has a knowledge base of programming plans, which are implementations of programming goals. One programming problem may have different goal decompositions. Figure 12 shows the simplified plan for the "Sentinel-Controlled Input Sequence goal." PROUST tries to recognise these plans, including erroneous plans, in the submitted code and reports bugs. Later, Sack proposed an improved system (PROGRAMCRITIC) based on MicroPROUST (a simpler version of PROUST) by simplifying its design and fixing its weaknesses, including the elimination of the distinction between goals and plans.

MENO-II is a predecessor to PROUST, which "failed miserably" [89] according to the authors. The tool uses a library of 18 common bug templates for simple programs containing a loop, derived from a set of student programs. The bug templates are linked to possible misconceptions, but are not problem-specific. The tool identifies plans in a student program, matches them against the bug templates (see Figure 13) and reports errors, misconceptions and possible solutions. This approach failed to recognise many mistakes in exercises, because the bug templates were too general and lacked information on how program components worked together to solve a particular problem. A better solution was needed to support the great variety in student programs.

Some of the IBD systems only work for a limited set of predefined exercises, such as the classic Rainfall problem for PROUST.

7.2.5 External Tools (EX). We have found a number of static analysis tools, for example CheckStyle[4] for checking code conventions, FindBugs [78][5] for finding bugs, and PMD[6] for detecting bad coding practices. These tools do not specifically focus on novice programmers and may produce

[3] findbugs.sourceforge.net.
[4] checkstyle.sourceforge.net.
[5] findbugs.sourceforge.net.
[6] pmd.github.io.

SENTINEL READ-PROCESS REPEAT PLAN

Constants: ?Stop
Variables: ?New
Template:
repeat
subgoal Input(?New)
subgoal Sentinel Guard(?New, ?Stop, ?∗)
until ?New = ?Stop

Fig. 12. Simplified plan in PROUST (image from Reference [87]).

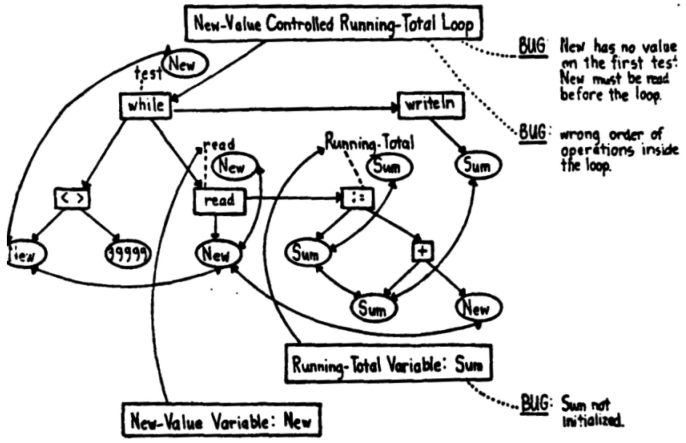


Fig. 13. Bugs located in a parse tree in MENO-II (image from Reference [163]).

output that is difficult to understand for beginners. The tools are often configured to provide a limited set of output messages so as not to overwhelm and confuse the learner.

7.3 Other Techniques

Of all tools, 27.7% use techniques that do not fit one of our labels, which are often AI techniques.

As an example, the ProPL tutor uses natural language processing to engage in a dialogue with the student to practice planning and program design. Human tutoring is a proven technique for effective learning. The tutor mimics the conversation that a human tutor would have with a student using natural language processing. ProPL uses a dialogue management system that requires a substantial amount of input to construct a tutor for a programming problem.

DATLAB employs machine learning techniques to classify student errors and generate corresponding feedback. The author uses a neural network to "learn relationships corresponding to trained error categories, and apply these relationships to unseen data" [117].

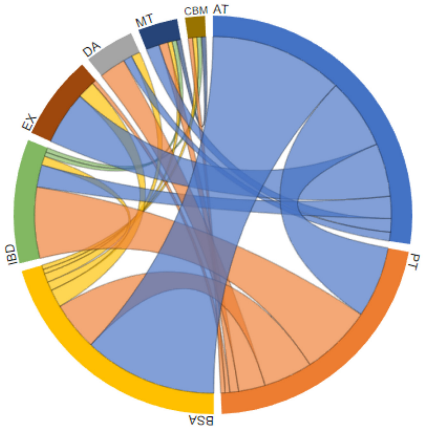
The COALA system uses fuzzy logic to analyse similarity to a model solution. The system calculates the score for a set of metrics (such as cyclomatic complexity, lines of code, number of parameters) for both the student program and model program, fuzzifies the scores, and calculates feedback on style issues and how to improve them using fuzzy rules such as:

```
IF (control complexity is low AND cyclomatic complexity is high)
THEN show message ('The solution can be done easily by reviewing
                    the conditions and deleting some bifurcation')
```

In addition, the system uses test cases to determine the correctness of the solution. The instructor has to provide a model program and a set of test cases but also the region values for each metric that are used to generate the fuzzy sets.

The authors of the Singh13 tool use program synthesis techniques to generate feedback on solutions for Python programming problems, such as [161]

```
The program requires 3 changes:
- In the return statement return deriv in line 5 , replace deriv by [0] .
- In the comparison expression (poly[e] == 0) in line 7, change (poly[e] == 0)
to False .
- In the expression range(0, len(poly)) in line 6, increment 0 by 1.
```



Combination	Count
Automated testing + Basic static analysis	30
Automated testing + Program transformations	16
Intension-based diagnosis + Program transformations	16
Automated testing + External tools	12
Program transformations + Basic static analysis	11
Data analysis + Program transformations	6

Fig. 14. Chord diagram showing the frequency of combinations of two techniques in the 101 tools through the width of a connection, omitting "Other." The table on the right shows all combinations that occur more than five times, also omitting "Other."

The problem author writes a model solution and an error model. The error model consists of a set of correction rules that solve a mistake together with appropriate feedback messages. All possible programs based on these rules applied to the student's solution are then searched to find the one that most closely matches the model solution. This is done by translating the program with the correction rules into a Sketch program. Sketch is a software synthesis tool that can complete a partial code implementation to make it behave like a given specification. The Sketch synthesiser finds the solution and feedback is generated based on the applied correction rules. A limitation of the tool is its inability to deal with student programs that have large conceptual errors.

The technique employed in Gulwani14 focuses on performance issues in student solutions. They observe that the percentage of solutions that are algorithmically inefficient is around 60% and can go up to 90% for their example problem of checking if two strings are anagrams of each other. In their solution, an instructor annotates a program by specifying key values that are computed during the execution of the program, thereby ignoring irrelevant implementation differences. A new language construct called "observe" is introduced to specify these values. The execution trace of a student program is compared to the traces of annotated programs and for the matching program an appropriate feedback message is shown. Although the instructor has a substantial task to annotate programs with different algorithmic approaches, the authors show that this is worth the effort for use in large-scale systems.

7.4 Combining Techniques

Figure 14 shows how often combinations of two techniques are used in one tool. Because our review excludes tools that only use automated testing (AT) and give feedback in the form of test case results, this technique is the one combined most with other techniques. AT is most often (in 30 tools) combined with basic static analysis (BSA). After that, combinations with program transformations (PT) are seen most often (in 16 tools), followed by combinations with external tools (EX) in 12 tools. Intention-based diagnosis (IBD) and PT are used in the same tool in 16 cases. BSA is also often combined with PT: We see this in 11 tools.

As a concrete example, the assessment technique of Vujošević-Janičić13 combines multiple techniques (testing, automated bug finding and control flow graph similarity) to overcome the

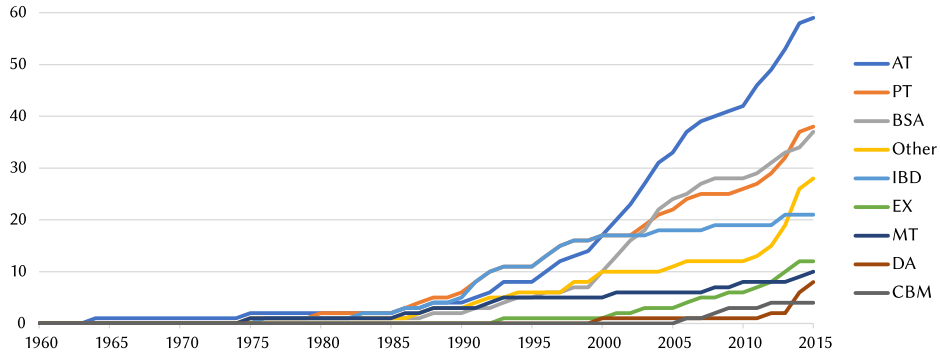


Fig. 15. Cumulative number of tools with technique over the years, based on the earliest paper on a tool from our data set. The legend has the same order as the endings of the lines in the chart.

shortcomings of only using a single technique. The authors state that software verification tools can pick up missing bugs not covered by a test case. Moreover, structural issues such as modularity cannot be intercepted by testing and verification only, so the similarity to a model program should also be incorporated in an assessment. In their solution, the C–code of students is translated into the LLVM intermediate language[7] and static verification is performed on the intermediate representation. The program is also translated into a control flow graph (CFG) and compared to the CFG of model programs, calculating the degree of similarity. The results are used to calculate a grade, but can also be used to give feedback on bugs and similarity to a model solution.

The ITS designed by Dadic11 uses different techniques for different types of students. The system uses model tracing to force "stoppers," students that give up quickly, to write a program in a predefined order. "Movers," students that will keep trying even when they are struggling, are allowed more freedom in their problem-solving process, for which the system uses a constraint-based technique to check their solution.

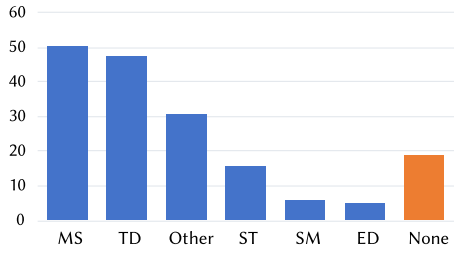
7.5 Trends

Figure 15 shows the techniques that have been used over the years. Intention based diagnosis emerged around the 1990s, but has been used less often in the 21st Century. Automated testing is most seen in the 2000s, but it remains popular in the 2010s. The same applies to basic static analysis. In the 2010s various new techniques, such as data analysis, make their appearance.

8 ADAPTABILITY (RQ3)

In this section, we describe the results of the third research question: "How can the tool be adapted by teachers, to create exercises and to influence the feedback?" To answer this question, we have categorised the different types of input that teachers can provide. We require that the tool does not need to be recompiled, and not too much effort or specialised knowledge is needed. Some tools enable authors to write complex error models or rules and constraints to specify correct solutions. However, we consider these inputs to be too specialised and time-consuming for a teacher who quickly wants to add a new exercise.

[7] www.llvm.org/.



		All (n=101)		Class 2 (n=24)		Class 3 (n=77)	
Input type		Count	%	Count	%	Count	%
MS	Model solutions	51	50.5	8	33.3	43	55.8
TD	Test data	48	47.5	4	16.7	44	57.1
Other		31	30.7	6	25.0	25	32.5
ST	Solution templates	16	15.8	7	29.2	9	11.7
SM	Student model	6	5.9	3	12.5	3	3.9
ED	Error data	5	5.0	0	0.0	5	6.5
None		19	18.8	9	37.5	10	13.0

Fig. 16. Number of tools (count) and percentage of tools with input type, for all tools, and subdivided by exercise class.

Figure 16 shows for each input type the percentage of tools that use it. We have found that model solutions are used most, in 50.5% of the tools. After that, 47.5% of the tools use test data. Of all tools, 15.8% offer solution templates, and in 5.0% error data can be specified. We have found the use of a student model for generating feedback on solutions in 5.9% of the tools. In 30.7% of the tools we have found other types of input. A total of 18.8% of the tools do not use any input type. The feedback these tools generate is often based on hard-coded knowledge bases (e.g., He94), or it is just too time-consuming to add an exercise (e.g., one person-week for Bridge).

Class 2 tools more often do not use any input. Class 3 tools use more test data and model solutions than class 2 tools. Class 2 tools use more solution templates and input from a student model. The next subsections expand on the different input types and explain how they are used.

8.1 Solution Templates (ST)

Solution templates are often used for class 2 exercises to restrict the student's freedom in solving a particular problem. An example is the ELP system shown in Figure 17, in which students fill in gaps (the white area in the middle) in a Java template with predefined code fragments (the yellow areas surrounding the white gap). Solution templates found in test-based assessment tools are often project skeletons, or an interface definition for a data structure that prescribes the names of functions, parameters, and return values.

8.2 Model Solutions (MS)

Correct solutions to a programming exercise are used as input in many tools. In dynamic analysis they are used for running test cases to generate the expected output. In this case a single correct solution will suffice. In static analysis the structure of a correct solution is compared to the structure of a student solution. To recognise more than one solution variant, some tools accept multiple solutions that each represent a different algorithm to solve the problem.

```
    //Read lower and upper limit
    lowerLimit =
         reader.readInt("lower limit: ");
    upperLimit =
         reader.readInt("upper limit: ");
```

```
    counter = lowerLimit;
    while(((counter <= upperLimit)== true)
             && (counter >=0))
    {
       writer.println("counter = " +
                                    counter);
       counter = counter + 1;
    }
```

```
 }
 public static void main(String[] args)
 {
    SafeCountBy1 tpo = new SafeCountBy1();
    tpo.run();
 }
```

Fig. 17. Fill-in-the-gap exercise in ELP (image fragment from Reference [176]).

```
if nnn ≧ 100 then
  begin error := true;
  outtext(∢ < No convergence after 100 calls. ∢);
  go to next problem
  end;
```

Fig. 18. Test code in NAUR64 (image from Reference [132]).

```
public final void testFactorial3() {
try {
   assertEquals(6, Factorial.factorial(3));
} catch (Exception e) {
   fail("Test failed for factorial(3)!");}}
```

Fig. 19. Unit test used in WEBWORK-JAG (image fromReference [59]).

8.3 Test Data (TD)

In older tools testing is done in scripts, for example, in NAUR64. Figure 18 shows a small fragment of its test code. Newer systems accept unit tests as input, such as in Figure 19, which shows one of the unit tests for the factorial function for WEBWORK-JAG. The WEBWORK-JAG system also processes Java reflection code to test the signature of methods. Other tools require the teacher to simply provide input/output pairs.

8.4 Error Data (ED)

A small number of tools accept input containing some form of error data. As an example, TALUS, a debugger for Lisp programs, uses both model solutions and buggy solutions. If a student program is matched with a buggy solution, then corresponding feedback addressing the misconception is presented to the student.

8.5 Other

In COURSEMARKER teachers can configure how much feedback should be given. Some tools let a teacher define custom feedback messages. In ASK-ELLE model solutions can be annotated with feedback messages [55]:

```
range x y = {-# FEEDBACK Note... #-}
        take (y-x+1) $ iterate (+1) x
```

These messages appear if the student asks for help at a specific stage during problem solving.

AUTOSTYLE provides a user interface for an instructor that visualises paths consisting of subsequent submissions and their differences. The instructor can influence the hints by adjusting thresholds so the changes between submissions become larger or smaller, and can mark hints as helpful or not, as shown in Figure 20.

We found a system that *generates* exercises instead of having instructors author them (SHIMIC), although the resulting exercises seem rather contrived ("Define private long integer method 'n' with integer argument named 'n,' and which returns: '1,022' if 'n' is '8,142'; '963' if 'n' is '1,261'; etc.").

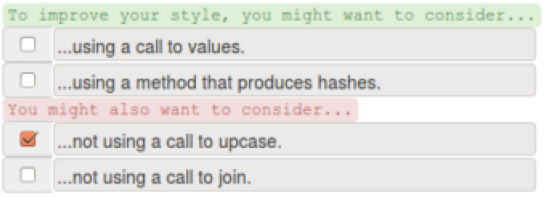
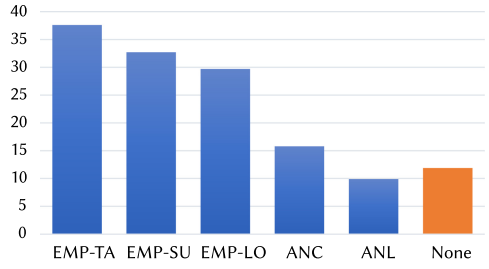


Fig. 20. Instructor UI for AutoStyle (image fragment from Reference [129]) .



Evaluation method		%
EMP-TA	Empirical – Technical analysis	37.6
EMP-SU	Empirical – Surveys	32.7
EMP-LO	Empirical – Learning outcome evaluations	29.7
ANC	Anecdotal assessment	15.8
ANL	Analytical assessment	9.9
None		11.9

Combination	%
1 method, not ANC	45.5
2 methods	16.8
3 methods	7.9
4 methods	2.0

Fig. 21. Percentage of tools (n = 101) with evaluation method and combinations of evaluation methods.

Testovid is a platform and language-independent tool, for which teachers have to write Apache Ant scripts to build student solutions and run a set of (testing, style checking) tools on the programs. Writing these scripts is a time investment, but they can be reused and adapted later for different exercises and languages.

In AutoTeach a model solution is gradually revealed (see Figure 8 in Section 6.4.2) The tool offers the teacher several options to customise the hints using "meta-commands," which are special directives embedded in comments. Using these commands the teacher can specify custom hint messages and control which parts of the code should (not) be revealed. The system includes a default revealing mechanism that starts with showing only the basic structure of the code (classes, functions) and ending at the level that almost shows all code except for instructions inside conditional statements and loops.

9 QUALITY (RQ4)

This section describes the results of the fourth research question: "What is known about the quality and effectiveness of the feedback or tool?" Figure 21 shows for each evaluation method the percentage of tools for which it has been used, and the number of evaluation methods used per tool. We have found that 15.8% of the tools we examined only provide anecdotal evidence on the success of a tool, and for 11.9% of the tools we have not found any evaluation at all. Of all tools, 9.9% have been assessed by an analytical method and 71.3% by some form of empirical assessment, of which technical analysis is the largest group with 37.6% of the tools. Not including anecdotal

assessment, the majority of the tools have been evaluated by one method (45.5%), 26.7% of the tools have been evaluated by at least two methods, 7.9% by three methods and 2.0% by four methods.

In the following subsections, we give some examples and observations regarding tool evaluation.

9.1 Analytical (ANL)

We found a small number of tools that are based on validated approaches, such as a learning theory. The LISP tutor, Grace, and JITS are based on the ACT-R cognitive architecture [6]. In ACT-R procedural knowledge is defined as a set of production rules that model human behaviour in solving particular problems. Chang00 is based on the completion strategy for learning programming by Van Merriënboer [180]. This strategy is based on exercises in which (incomplete) model programs written by an expert should be completed, extended, or modified by a novice programmer.

9.2 Empirical Assessment

Empirical assessment analyses qualitative data or quantitative data. We distinguish three types.

9.2.1 Looking at the Learning Outcome (EMP-LO). The nature of these experiments varies greatly. For example, ProPL was evaluated in an experiment with 25 students. The students were either in the control group that used a simple, alternative learning strategy, or in the group that used ProPL. Students both did a pre-test and a post-test to assess the changes in the scores. Some tools have used a less extensive method, for instance by omitting a control group, and comparing the pass rates from the year the tool was used against previous years. The test group size also varies greatly.

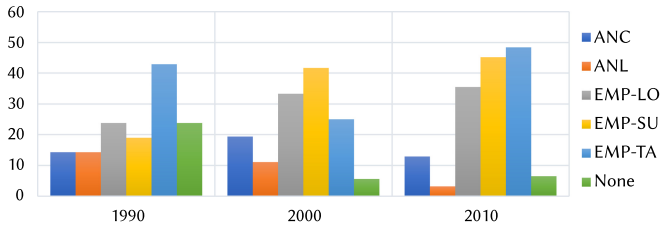
Evaluating the programming game Code Hunt was done by tracking the activity of 407 users over two weeks. Some users got all hints, some users got occasional hints, and some users never got any hints. The results show that the users who got hints played longer and won more levels than the users who did not get any hints. In addition, users who got occasional hints played longer than users that always got hints.

9.2.2 Student and Teacher Surveys (EMP-SU). We have found that the number of responses in some cases is very low, or is not even mentioned. Some papers mention a survey but do not show an analysis of the results, in which case we do not assign this label.

9.2.3 Technical Analysis (EMP-TA). SIPLeS-II was assessed using a set of 525 student programs, measuring the number of correctly recognised solutions, the time needed for the analysis, and a comparison to the analysis of a human tutor. In some cases, this type of analysis is done for a large number of programs, only counting the number of recognised programs. In other cases, researchers thoroughly analyse the content of generated hints, often for a smaller set of programs because of the large amount of work involved.

The authors of Jin12 analysed 200 student submissions for one simple programming problem ("calculate the pay for mowing the lawn around a house"). In a first experiment the 37 correct solutions from this set were used to generate hints for 16 randomly selected correct submissions. The hints were appropriate for 14 of these 16 submissions (87.5%). In their second experiment, the authors manually selected a similar correct solution, "which was not necessarily the best match" for 15 randomly selected incorrect submissions. The generated hints based on their differences were meaningful for 10 (66.6%) of the 15 incorrect submissions. The authors propose concrete ways to increase this percentage.

The technique from Jin12 was used in the Jin14 tutor, which has a "guided-planning" and an "assisted-coding" component. The tutor was evaluated by comparing it to three other variants: "coding-only," "planning-only," and alternating between variants. They conducted an experiment



		1990	2000	2010
Evaluation method		%	%	%
ANC	Anecdotal assessment	14.3	19.4	12.9
ANL	Analytical assessment	14.3	11.1	3.2
EMP-LO	Empirical - Learning outcome evaluations	23.8	33.3	35.5
EMP-SU	Empirical - Surveys	19.0	41.7	45.2
EMP-TA	Empirical - Technical analysis	42.9	25.0	48.4
None		23.8	5.6	6.5

Fig. 22. Percentage of tools with evaluation method in the 1990s (n = 21), 2000s (n = 36), and 2010s (n = 31). The legend and table have the same order as the bars in the chart.

with 85 students who did a pre-test, worked with the tutor for some time, and then completed a post-test. The group that did the "planning-coding" variant had the largest increase in learning.

9.2.4 Other. Another way to evaluate a tool is to compare it to other related tools based on a set of criteria or functions (e.g., Java Sensei). The feedback provided by the Fit Java Tutor was assessed for their appropriateness by experts. The authors of this tool also did a "Wizard of Oz" experiment to assess if the example-based feedback helped students to achieve higher scores on their work. Instead of the ITS, a human expert provided the feedback that students requested. Some tools were evaluated as part of a course that used the tool, or in the context of a specific didactic method, making it more difficult to isolate and measure the effect of the tool itself.

9.3 Trends

Figure 22 shows how the distribution of evaluation methods changes over the last three decades. The use of surveys shows a steady increase from 19.0% in the 1990s, 41.7% in the 2000s, and 45.2% in the 2010s. Despite the relapse to 25.0% in the 2000s, technical analysis is still the method seen most in this decade with 48.4% in the 2010s, closely followed by surveys and learning outcome evaluations. The number of tools for which we have not found any type of evaluation drops from 23.8% to around 6%, but the percentage of tools for which we only found anecdotal evidence remains nearly the same. It should be noted, however, that some evaluations of tools from this decade are still to be published, so this figure only gives a general impression.

10 DISCUSSION

In this review, we intend to find an answer to our research question concerning feedback generation for programming exercises: "What is the nature of the feedback, how is it generated, can a teacher adapt the feedback, and what can we say about its quality and effect?" In this section, we take a closer look at the answers we have found to the four sub-questions, discuss the relation between these answers, and give some observations and recommendations that follow from our review.

10.1 Feedback Types

Looking at the type of feedback given by tools, which we investigated in the first research question, we have found that feedback about mistakes is the largest type used in tools. In the first iteration of our review, we found that feedback on test failures was the largest subtype [96]. After deciding to omit purely test-based tools in this final review, this subtype was still the second largest. Generating feedback based on tests is a useful way to point out errors in student solutions and emphasizes the importance of testing to students. It is therefore a valuable technique and relatively easy to implement using existing test frameworks. Most tools that use automated testing support class 3 exercises, because black-box testing does not require using a specific algorithm or design process. The only aspect that matters is whether the output meets the requirements of the exercise. However, only giving test-based feedback will not in all cases help a student to fix an incorrect program.

The largest feedback subtype from our review is solution error feedback. This type of feedback can be implemented with varying depth and detail. To really help a student, just pointing at an error may not help. Feedback on how to proceed is necessary to both fix problems and to progress towards a solution when stuck. This type of feedback is mostly seen in the form of error correction hints, and much less as hints on task-processing steps or on program improvements. This is especially the case for class 3 tools supporting exercises that can be solved by multiple (variants of) strategies. Class 2 tools provide more procedural support, but a disadvantage is that they do not support alternative solution strategies and may restrict a student in his or her problem-solving process. Finally, the very low percentage of tools that give code examples based on the student's actions is unfortunate, because studying examples has proven to be an effective way of learning. Looking at the changes over the decades, feedback on test failures and solution errors remain the most common types. However, the diversity of feedback types in the current century has increased, which is a positive development.

Many of the tools we have investigated are AA tools, which are often used for marking large numbers of solutions. If marking is the only purpose, then one could conclude that more elaborate feedback is not necessary. However, if we want our students to learn from their mistakes, then a single mark or a basic list of errors only is not sufficient. Moreover, we have noticed that many authors of AA tools claim that the intention of the feedback their tool generates is student learning.

Suzuki et al. [170] analysed posts on a programming discussion forum to identify the different types of hints that teachers give. The authors found 10 types, 5 of which could be generated using program synthesis techniques: transformation hints (how to correct mistakes), location hints (where to correct mistakes), data hints (the expected value or type of a variable at some location), behaviour hints (describing dynamic program behaviour), and example hints (explaining how code should work by showing input and output examples). The authors found that hints teachers give often focus on *why* the student's code failed and that teachers generally do not provide exact fixes.

There is a vast amount of studies that identify and classify difficulties and errors that novice programmers encounter in their learning process. Qian and Lehman [145] have conducted a literature review of such studies of student difficulties in introductory programming. The authors distinguish between difficulties in syntactic knowledge, conceptual knowledge (not knowing how a programming construct works or how code executes), and strategic knowledge (using code to solve problems), and they describe several approaches and tools to support students in dealing with these difficulties. Considering our classification of feedback, misconceptions regarding syntactic knowledge can be addressed by compiler error feedback. Conceptual knowledge can be addressed by the two KC (knowledge about concepts) feedback types: explanations on subject matter and examples illustrating concepts. Strategic knowledge can be addressed by feedback on solution errors,

all feedback on knowledge about how to proceed, and knowledge on task constraints. Feedback on test failures, style, and performance can also help.

Offering more variety in feedback type in one tool than is currently the case and carefully considering the level of detail in a generated feedback message would contribute to giving more effective feedback, similar to the feedback of a human tutor.

10.2 Feedback Generation Techniques

By answering our second research question, we came across many techniques for generating automated feedback, which we can relate to the different feedback content types. Well-known ITS techniques such as model tracing and constraint checking are used in some tools, but not on a very large scale. Model tracing is strongly related to hints in the "knowledge on how to proceed" category, often producing next-step hints. Constraint checking and intention-based diagnosis mainly produce hints on solution errors and/or error correction. Automated testing is often combined with static analysis of the abstract syntax tree, sometimes incorporating transformations to simplify its structure or to ignore irrelevant details. We observe that automated testing generates test-based feedback. However, it is also used as a mechanism for recognising a solution as correct or incorrect before further analysing a program using another technique. For generating style-related hints, basic static analysis or external tools (that might also perform static analysis) are used.

An upcoming trend is the use of data-driven technology to base feedback on historical student data, such as paths students have taken and programming mistakes they have made. Hint generation with data-driven techniques produces feedback on solution errors, correcting these errors, taking a next step and providing the student with related examples. Feedback generation based on recognising plans and errors from a knowledge base is less popular nowadays. Data-driven techniques, however, require the presence of large sets of data, complicating the authoring of new exercises. We must also find an answer to the question if students learn the right problem-solving strategies if hints are only based on other students' past behaviour. Perhaps combining the successful techniques of the past with new (data-driven) enhancements will bring new opportunities.

10.3 Tool Adjustability

The 2014 working group report of the ITiCSE conference by Brusilovsky et al. [26] discusses the adoption of "smart learning content" (SLC) in computer science courses on a larger scale. SLC tools are interactive tools offering, for instance, visualisation, simulation, automated assessment, coding support, or problem-solving support, in which feedback plays a prominent role to help students in their learning. One of the main problems the authors found among teachers is not being able to customise tools to their own needs, and that tools do not match their own teaching methods. The report proposes an architecture that promotes flexible integration and customisation of SLC tools.

In our review, we found that tools use various dynamic and static analysis techniques. More sophisticated techniques, such as model tracing and intention-based diagnosis, appear to complicate adding new exercises and adjusting the tool. However, the question whether or not a tool can be adapted easily is difficult to answer and depends on the amount and complexity of the input. We have found that very few papers explicitly describe this or even address the role of the teacher. In the latter case, we assume that there is no such role and the tool can only be adjusted by the developers. When a publication does describe how an exercise can be added, it is often not clear how difficult this is. Some publications mention the amount of time necessary to add an exercise, such as one person-week for Bridge. We conclude that teachers cannot easily adapt tools to their own needs, except for test-based AA systems.

10.4 Tool Evaluation

To answer the last question, we have investigated how tools are evaluated. Most tools provide at least some form of evaluation, although for 27.7% of the tools we could only find anecdotal evidence, or none at all. The evaluation of a tool may not be directly related to the quality of the feedback, so the results only give a general idea of how much attention was spent on evaluation. The many different evaluation methods make it difficult to assess the effectiveness of feedback. Moreover, the quality (e.g., presence of control groups, pre- and post-tests, group size) of empirical assessment varies greatly. Finally, the description of the method and results often lacks clarity and detail.

Gross and Powers [61] provide a rubric for evaluating the quality of empirical tool assessments and have applied this rubric to the evaluation of a small set of tools. Collecting data for this rubric would provide us with more information, but the effort is beyond the scope of this review. Just as Gross and Powers conclude, the lack of information on assessment greatly complicates this task. Pettit and Prather also endorse the need for developers of AA systems to work together instead of creating tools in isolation and to pay more attention to evaluating their effectiveness [140].

In the future, it would be interesting to compare tools that give different types of feedback, to assess the effectiveness of different (combinations of) feedback types. Furthermore, extensive technical analyses are needed to verify to what extent a tool can correctly recognise (in)correct solutions and generate appropriate hints and for which subset of exercises.

10.5 Classifying Feedback

In this section, we compare our classification of feedback to another classification from a recent paper. This paper by Le provides "a classification of adaptive feedback in educational systems for programming" [107]. The author describes "adaptive feedback" as feedback specific for the actions of an individual student, possibly combined with information from a student model. The classification consists of five main feedback types: "yes-no," "syntax," "semantic," "layout," and "quality." Although the author states that these types are based on generic classifications of feedback types (including Narciss), we cannot clearly derive from the paper how the author's classification relates to these general classifications. If we compare these types to ours, then yes-no feedback relates to Narciss' "knowledge of result/response," which we do not include in our review. Syntax feedback relates to our "compiler errors" type. Le distinguishes two levels in the semantic feedback category: intention-based (feedback on the solution strategy for solving a particular task) and code-based (feedback on coding errors with respect to a particular task), which are related to "solution errors," both subtypes of "knowledge about how to proceed," and "hints on task-processing rules." Layout feedback corresponds to "style issues" and quality feedback to "performance issues." Although we do see similarities between both categorisations, ours differs in three ways: it resembles general feedback types at the top level, it includes a broader range of types, and it is slightly more fine-grained.

10.6 Threats to Validity

There are a number of factors that could influence the validity of our review during the search for publications and the coding of the tools. We could have missed some papers, because their authors use terms different from our database search string. We take this into account by inspecting references, but papers with few references might not have been found this way. For this reason, we could have missed some recent papers that typically do not have many references yet.

The substantial amount of work involved in this review, and therefore the time it took to execute the coding, may have had an effect on the interpretation of labels, in particular the borderline cases.

In some cases, we could not find clear answers to our research questions and had to speculate on what the correct label was. However, in these cases a second author was often consulted.

For answering RQ4, we might not have found some evaluations of recent tools that were conducted at a later date, because we have only included papers up to 2015. Validation of a tool is usually done by its authors, but in some cases may be performed by other authors. We did not do a search of these "external" evaluations.

11 CONCLUSION

We have analysed and categorised the feedback generation in 101 tools for learning programming, selected from 17 earlier reviews and two databases. We have reported findings on the relation of feedback content, technique and adaptability. We have found that, in general, the feedback that tools generate is not very diverse, and mainly focused on identifying mistakes. In tools that support class 3 exercises, test-based feedback is most common and very few of these tools give feedback with "knowledge on how to proceed." Class 2 tools give more feedback on fixing mistakes and taking a next step, but at the cost of not recognising alternative strategies. Offering multiple feedback types in one tool and considering the level of detail would contribute to giving more effective feedback, similar to the feedback of a human tutor. We already see more diversity of feedback types in tools developed in the 21st Century.

We have found many different techniques used to generate feedback, both general and specific to the programming domain, often strongly related to a specific type of feedback. The upcoming trend of data-driven tutors shows promising results but also poses questions for which we need to find answers regarding the effectiveness of the hints they produce. Requiring large data sets also complicates the authoring of new exercises. We observe that teachers cannot easily adapt tools to their own needs, except for providing test data as input for a tool. Finally, the quality of the evaluation of tools varies greatly and is still an area for improvement. Better technical analyses should make it clearer what the features and limitations of a tool are, and better experiments that measure learning should give more insight into the effectiveness of a tool.

Automated feedback for programming exercises has been an active field of research for many decades. Generating effective feedback messages and making it easy for teachers to adapt tools to their own needs will remain a challenge, although many useful techniques have already been employed in practice. New technologies combined with techniques from this review that have proven to be effective will undoubtedly help new students wanting to learn how to program even better in the future.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable comments and suggestions.

REFERENCES

[1] Anne Adam and Jean-Pierre Laurent. 1980. LAURA, a system to debug student programs. *Artific. Intell.* 15, 1–2 (1980), 75–122.

[2] Kirsti Ala-Mutka. 2005. A survey of automated assessment approaches for programming assignments. *Comput. Sci. Edu.* 15, 2 (2005), 83–102.

[3] Kirsti Ala-Mutka and Hannu Matti Järvinen. 2004. Assessment process for programming assignments. In *Proceedings of the IEEE Conference on Advanced Learning Technologies.* 181–185.

[4] Vincent Aleven, Bruce Mclaren, Jonathan Sewall, and Kenneth Koedinger. 2009. A new paradigm for intelligent tutoring systems: Example-tracing tutors. *Int. J. Artific. Intell. Edu.* 19 (2009), 105–154.

[5] Dean Allemang. 1991. Using functional models in automatic debugging. *IEEE Expert* 6, 6 (1991), 13–18.

[6] John R. Anderson. 1983. *The Architecture of Cognition.* Lawrence Erlbaum Associates, Inc.

[7] John R. Anderson and Edward Skwarecki. 1986. The automated tutoring of introductory computer programming. *Commun. ACM* 29, 9 (1986), 842–849.

[8] Paolo Antonucci. 2014. *AutoTeach: Incremental Hints For Programming Exercises*. Master's thesis. ETH Zurich.
[9] Paolo Antonucci, Christian Estler, Đurica Nikolić, Marco Piccioni, and Bertrand Meyer. 2015. An incremental hint system for automated programming assignments. In *Innovat. Technol. Comput. Sci. Edu.* 320–325.
[10] David Arnow and Oleg Barshay. 1999. WebToTeach: An interactive focused programming exercise system. In *Proceedings of the Frontiers in Education Conference*, Vol. 1. 39–44.
[11] Avron Barr and Marian Beard. 1976. An instructional interpreter for basic. *ACM SIGCSE Bull.* 8, 1 (1976), 325–334.
[12] Avron Barr, Marian Beard, and Richard C. Atkinson. 1975. A rationale and description of a CAI program to teach the BASIC programming language. *Instruct. Sci.* 4, 1 (1975), 1–31.
[13] Avron Barr, Marian Beard, and Richard C. Atkinson. 1976. The computer as a tutorial laboratory: The Stanford BIP project. *Int. J. Man-Mach. Studies* 8, 5 (1976), 567–596.
[14] María Lucía Barrón-Estrada, Ramón Zatarain-Cabada, Francisco González Hernández, Raúl Oramas Bustillos, and Carlos A. Reyes-García. 2015. An affective and cognitive tutoring system for learning programming. In *Advances in Artificial Intelligence and Its Applications*. Vol. 9414, LNCS. 171–182.
[15] Christoph Beierle, Marija Kulaš, and Manfred Widera. 2003. Automatic analysis of programming assignments. In *DeLFI: Die 1. e-Learning Fachtagung Informatik*. 144–153.
[16] Christoph Beierle, Marija Kulaš, and Manfred Widera. 2004. Partial specifications of program properties. In *Proceedings of the International Workshop on Teaching Logic Programming*. 18–34.
[17] Steve Benford, Edmund Burke, and Eric Foxley. 1993. Learning to construct quality software with the ceilidh system. *Softw. Qual. J.* 2, 3 (1993), 177–197.
[18] Steve Benford, Edmund Burke, Eric Foxley, Neil Gutteridge, and Abdullah Mohd Zin. 1993. Early experiences of computer-aided assessment and administration when teaching computer programming. *Res. Learn. Technol.* 1, 2 (1993), 55–70.
[19] Steve Benford, Edmund Burke, Eric Foxley, and Colin Higgins. 1995. The ceilidh system for the automatic grading of students on programming courses. In *Proceedings of the ACM Southeast Conference*. 176–182.
[20] Jens Bennedsen and Michael E. Caspersen. 2007. Failure rates in introductory programming. *ACM SIGCSE Bull.* 39, 2 (2007), 32–36.
[21] Michael Blumenstein, Steve Green, Shoshana Fogelman, Ann Nguyen, and Vallipuram Muthukkumarasamy. 2008. Performance analysis of GAME: A generic automated marking environment. *Comput. Edu.* 50 (2008), 1203–1216.
[22] Michael Blumenstein, Steve Green, Ann Nguyen, and Vallipuram Muthukkumarasamy. 2004. GAME: A generic automated marking environment for programming assessment. In *Proceedings of the Conference on Information Technology: Coding and Computing*, Vol. 1. 212–216.
[23] Jeffrey G. Bonar and Robert Cunningham. 1988. *Bridge: Intelligent Tutoring with Intermediate Representations*. Technical Report. Carnegie Mellon University, University of Pittsburgh.
[24] David Boud and Elizabeth Molloy (Eds.). 2012. *Feedback in Higher and Professional Education: Understanding it and Doing it Well*. Routledge.
[25] Peter Brusilovsky. 1992. Intelligent tutor, environment and manual for introductory programming. *Innovat. Edu. Train. Int.* 29, 1 (1992), 26–34.
[26] Peter Brusilovsky, Stephen Edwards, Amruth Kumar, Lauri Malmi, Luciana Benotti, Duane Buck, Petri Ihantola, Rikki Prince, Teemu Sirkiä, Sergey Sosnovsky et al. 2014. Increasing adoption of smart learning content for computer science education. In *Proceedings of the Working Group Reports of Innovation and Technology in Computer Science Education*. 31–57.
[27] Peter Brusilovsky and Gerhard Weber. 1996. Collaborative example selection in an intelligent example-based programming environment. In *Proceedings of the Conference on Learning Sciences*. 357–362.
[28] Julio C. Caiza and Jose M. Del Alamo. 2013. Programming assignments automatic grading: Review of tools and implementations. In *Proceedings of the International Technology, Education and Development Conference*. 5691–5700.
[29] Michael E. Caspersen and Jens Bennedsen. 2007. Instructional design of a programming course: A learning theoretic approach. In *Proceedings of the Workshop on Computing Education Research*. 111–122.
[30] Kuo En Chang, Bea Chu Chiao, Sei Wang Chen, and Rong Shue Hsiao. 2000. A programming learning system for beginners—A completion strategy approach 2000. *IEEE Trans. Edu.* 43, 2 (2000), 211–220.
[31] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. 2003. On automated grading of programming assignments in an academic institution. *Comput. Edu.* 41, 2 (2003), 121–131.
[32] Yam San Chee. 1995. Cognitive apprenticeship and its application to the teaching of Smalltalk in a multimedia interactive learning environment. *Instruction. Sci.* 23, 1-3 (1995), 133–161.
[33] Koen Claessen and John Hughes. 2011. QuickCheck: A lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices* 46, 4 (2011), 53–64.
[34] Albert T. Corbett and John R. Anderson. 1993. Student modeling in an intelligent programming tutor. In *Cognitive Models and Intelligent Environments for Learning Programming*. Vol. 111. Springer, 135–144.

[35] Albert T. Corbett and John R. Anderson. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Model. User-Adapt. Interact.* 4, 4 (1994), 253–278.
[36] Albert T. Corbett and John R. Anderson. 2001. Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 245–252.
[37] Albert T. Corbett, John R. Anderson, and Eric J. Patterson. 1990. Student modeling and tutoring flexibility in the lisp intelligent tutoring system. In *Intelligent Tutoring Systems*. Ablex, 83–106.
[38] Tonci Dadic. 2011. Intelligent tutoring system for learning programming. In *Intelligent Tutoring Systems in E-Learning Environments*. IGI Global, 166–186.
[39] Tonci Dadic, Slavomir Stankov, and Marko Rosic. 2008. Meaningful learning in the tutoring system for programming. In *Proceedings of the Conference on Information Technology Interfaces*. 483–488.
[40] Ronald Lee Danielson. 1975. *Pattie: An Automated Tutor for Top-down Programming*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.
[41] Leliane Nunes de Barros and Karina Valdivia Delgado. 2006. Model based diagnosis of student programs. In *Proceedings of the Monet Workshop on Model-Based System at ECAI*.
[42] Draylson M. De Souza, Seiji Isotani, and Ellen F. Barbosa. 2015. Teaching novice programmers using ProgTest. *Int. J. Knowl. Learn.* 10, 1 (2015), 60–77.
[43] Draylson M. De Souza, José Carlos Maldonado, and Ellen F. Barbosa. 2011. ProgTest: An environment for the submission and evaluation of programming assignments based on testing activities. In *Proceedings of the IEEE Conference on Software Engineering Education and Training*. 1–10.
[44] Draylson M. De Souza, Bruno H. Oliveira, José C. Maldonado, Simone R. S. Souza, and Ellen F. Barbosa. 2014. Towards the use of an automatic assessment system in the teaching of software testing. In *Proceedings of the Frontiers in Education Conference*. 1–8.
[45] Fadi P. Deek, Ki-Wang Ho, and Haider Ramadhan. 2000. A critical analysis and evaluation of web-based environments for program development. *Internet Higher Edu.* 3, 4 (2000), 223–269.
[46] Fadi P. Deek and James A. McHugh. 1998. A survey and critical analysis of tools for learning programming. *Comput. Sci. Edu.* 8, 2 (1998), 130–178.
[47] Christopher Douce, David Livingstone, and James Orwell. 2005. Automatic test-based assessment of programming: A review. *J. Edu. Res. Comput.* 5, 3 (2005).
[48] Stephen H. Edwards. 2003. Improving student performance by evaluating how well students test their own programs. *J. Edu. Res. Comput.* 3, 3 (2003), 1–24.
[49] Stephen H. Edwards and Manuel A. Pérez-Quiñones. 2007. Experiences using test-driven development with an automated grader. *J. Comput. Sci. Colleges* 22, 3 (2007), 44–50.
[50] John English and Tammy English. 2015. Experiences of using automated assessment in computer science courses. *J. Info. Technol. Edu.: Innovat. Pract.* 14 (2015), 237–254.
[51] Gregor Fischer and Jürgen Wolff von Gudenberg. 2006. Improving the quality of programming education by online assessment. In *Proceedings of the Symposium on Principles and Practice of Programming in Java*. 208–211.
[52] Eric Foxley and Colin A. Higgins. 2001. The CourseMaster CBA system: Improvements over ceilidh improvements over ceilidh. In *Proceedings of the CAA Conference*.
[53] Timothy S. Gegg-Harrison. 1993. *Exploiting Program Schemata in a Prolog Tutoring System*. Ph.D. Dissertation. Duke University.
[54] Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. 2010. Using strategies for assessment of programming exercises. In *Proceedings of the SIGCSE Technical Symposium on Computer Science Education*. 441–445.
[55] Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. 2012. An interactive functional programming tutor. In *Innovat. Technol. Comput. Sci. Edu.* 250–255.
[56] Moumita Ghosh, Brijesh Kumar Verma, and Anne T. Nguyen. 2002. An automatic assessment marking and plagiarism detection. In *Proceedings of the Conference on Information Technology and Applications*.
[57] Michael Goedicke, Michael Striewe, and Moritz Balz. 2008. *Computer Aided Assessments and Programming Exercises with JACK*. Technical Report 28. ICB, University Duisburg-Essen.
[58] Mercedes Gómez-Albarrán. 2005. The teaching and learning of programming: A survey of supporting software tools. *Comput. J.* 48, 2 (2005), 130–144.
[59] Olly Gotel, Christelle Scharff, and Andrew Wildenberg. 2008. Teaching software quality assurance by encouraging student contributions to an open source web-based system for the assessment of programming assignments. *ACM SIGCSE Bull.* 40, 3 (2008), 214–218.
[60] Olly Gotel, Christelle Scharff, Andrew Wildenberg, Mamadou Bousso, Chim Bunthoeurn, Phal Des, Vidya Kulkarni, Srisupa Palakvangsa Na Ayudhya, Cheikh Sarr, and Thanwadee Sunetnanta. 2008. Global perceptions on the use of WeBWorK as an online tutor for computer science. In *Proceedings of the Frontiers in Education Conference*. 5–10.

[61] Paul Gross and Kris Powers. 2005. Evaluating assessments of novice programming environments. In *Proceedings of the International Workshop on Computing Education Research*. 99–110.
[62] Sebastian Gross, Bassam Mokbel, Barbara Hammer, and Niels Pinkwart. 2015. Learning feedback in intelligent tutoring systems. *Künstliche Intelligenz* 29, 4 (2015), 413–418.
[63] Sebastian Gross, Bassam Mokbel, Benjamin Paassen, Barbara Hammer, and Niels Pinkwart. 2014. Example-based feedback provision using structured solution spaces. *Int. J. Learn. Technol.* 9, 3 (2014), 248–280.
[64] Sebastian Gross and Niels Pinkwart. 2015. Towards an integrative learning environment for java programming. In *Proceedings of the IEEE Conference on Advanced Learning Technologies*. 24–28.
[65] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2014. Feedback generation for performance problems in introductory programming assignments. In *Proceedings of the SIGSOFT International Symposium on Foundations of Software Engineering*. New York, New York, USA, 41–51.
[66] Mark Guzdial. 2004. Programming environments for novices. In *Computer Science Education Research*, Sally Fincher and Marian Petre (Eds.). CRC Press, 127–154.
[67] Budi Hartanto. 2014. *Incorporating Anchored Learning in a C# Intelligent Tutoring System*. Ph.D. Dissertation. Queensland University of Technology.
[68] Budi Hartanto and Jim Reye. 2013. CSTutor: An intelligent tutoring system that supports natural learning. In *Proceedings of the Conference on Computer Science Education Innovation and Technology*. 19–26.
[69] Helen M. Hasan. 1988. Assessment of student programming assignments in COBOL. *Edu. Comput.* 4 (1988), 99–107.
[70] John Hattie and Helen Timperley. 2007. The power of feedback. *Rev. Edu. Res.* 77, 1 (2007), 81–112.
[71] Yu He, Mitsuru Ikeda, and Riichiro Mizoguchi. 1994. Helping novice programmers bridge the conceptual gap. In *Proceedings of the Conference on Expert Systems for Development*. 192–197.
[72] Michael T. Helmick. 2007. Interface-based programming assignments and automatic grading of Java programs. *ACM SIGCSE Bull.* 39, 3 (2007), 63–67.
[73] Colin A. Higgins, Geoffrey Gray, Pavlos Symeonidis, and Athanasios Tsintsifas. 2005. Automated assessment and experiences of teaching programming. *J. Edu. Res. Comput.* 5, 3 (2005).
[74] Colin A. Higgins and Fatima Z. Mansouri. 2000. PRAM: A courseware system for the automatic assessment of AI programs. In *Innovative Teaching and Learning*. Vol. 1. Springer, 311–329.
[75] Colin A. Higgins, Pavlos Symeonidis, and Athanasios Tsintsifas. 2002. The marking system for CourseMaster. *ACM SIGCSE Bull.* 34, 3 (2002), 46–50.
[76] Jay Holland, Antonija Mitrovic, and Brent Martin. 2009. J-LATTE: A constraint-based tutor for Java. In *Proceedings of the Conference on Computers in Education*. 142–146.
[77] Jun Hong. 2004. Guided programming and automated error analysis in an intelligent Prolog tutor. *Int. J. Hum.-Comput. Studies* 61, 4 (2004), 505–534.
[78] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *ACM SIGPLAN Notices* 39, 12 (2004), 92–106.
[79] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the Koli Calling International Conference on Computing Education Research*. 86–93.
[80] Carlo Innocenti, Claudio Massucco, Donatella Persico, and Luigi Sarti. 1991. Ugo: An intelligent tutoring system for prolog. In *Proceedings of the PEG Conference on Knowledge Based Environments for Teaching and Learning*. 322–329.
[81] David Jackson. 1996. A software system for grading student computer programs. *Comput. Edu.* 27, 3–4 (1996), 171–180.
[82] David Jackson. 2000. A semi-automated approach to online assessment. *ACM SIGCSE Bull.* 32, 3 (2000), 164–167.
[83] David Jackson and Michelle Usher. 1997. Grading student programs using ASSYST. *ACM SIGCSE Bull.* 29, 1 (1997), 335–339.
[84] Johan Jeuring, L. Thomas van Binsbergen, Alex Gerdes, and Bastiaan Heeren. 2014. Model solutions and properties for diagnosing student programs in Ask-Elle. In *Proceedings of the Computer Science Education Research Conference*. 31–40.
[85] Wei Jin, Tiffany Barnes, and John Stamper. 2012. Program representation for automatic hint generation for a data-driven novice programming tutor. In *Intelligent Tutoring Systems*. Springer, 304–309.
[86] Wei Jin, Albert Corbett, Will Lloyd, Lewis Baumstark, and Christine Rolka. 2014. Evaluation of guided-planning and assisted-coding with task relevant dynamic hinting. In *Intelligent Tutoring Systems*. Springer, 318–328.
[87] W. Lewis Johnson. 1990. Understanding and debugging novice programs. *Artific. Intell.* 42, 1 (1990), 51–97.
[88] W. Lewis Johnson and Elliot Soloway. 1984. Intention-based diagnosis of novice programming errors. In *Proceedings of the AAAI Conference*. 162–168.
[89] W. Lewis Johnson and Elliot Soloway. 1985. PROUST: Knowledge-based program understanding. *IEEE Trans. Softw. Eng.* 11, 3 (1985), 267–275.
[90] Joint Task Force on Computing Curricula, ACM and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM.

[91] Francisco Jurado, Miguel Redondo, and Manuel Ortega. 2012. Using fuzzy logic applied to software metrics and test cases to assess programming assignments and give advice. *J. Netw. Comput. Appl.* 35, 2 (2012), 695–712.
[92] Francisco Jurado, Miguel Redondo, and Manuel Ortega. 2014. eLearning standards and automatic assessment in a distributed eclipse based environment for learning computer programming. *Comput. Appl. Eng. Edu.* 22, 4 (2014), 774–787.
[93] Sokratis Karkalas and Sergio Gutierrez-Santos. 2014. Enhanced javascript learning using code quality tools and a rule-based system in the FLIP exploratory learning environment. In *Proceedings of the IEEE Conference on Advanced Learning Technologies*. 84–88.
[94] Caitlin Kelleher and Randy Pausch. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *Comput. Surveys* 37, 2 (2005), 83–137.
[95] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2014. Strategy-based feedback in a programming tutor. In *Proceedings of the Computer Science Education Research Conference*. 43–54.
[96] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a systematic review of automated feedback generation for programming exercises. In *Innovation and Technology in Computer Science Education*. ACM, 41–46.
[97] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. *Towards a Systematic Review of Automated Feedback Generation for Programming Exercises—Extended Version*. Technical Report UU-CS-2016-001.
[98] Seon-Man Kim and Jin H. Kim. 1998. A hybrid approach for program understanding based on graph parsing and expectation-driven analysis. *Appl. Artific. Intell.* 12, 6 (1998), 521–546.
[99] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report EBSE-2007-01.
[100] Hikyoo Koh and Daniel Ming-Jen Wu. 1988. Goal-directed semantic tutor. In *Proceedings of the Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*. 171–176.
[101] Carsten Köllmann and Michael Goedicke. 2006. Automation of java code analysis for programming exercises. In *Proceedings of the Workshop on Graph Based Tools, Electronic Communications of the EASST*, Vol. 1. 1–12.
[102] Carsten Köllmann and Michael Goedicke. 2008. A specification language for static analysis of student exercises. In *Proceedings of the Conference on Automated Software Engineering*. 355–358.
[103] Utku Kose and Omer Deperlioglu. 2012. Intelligent learning environments within blended learning for ensuring effective C programming course. *Int. J. Artific. Intell. Appl.* 3, 1 (2012), 105–124.
[104] Angelo Kyrilov and David C. Noelle. 2015. Binary instant feedback on programming exercises can reduce student engagement and promote cheating. In *Proceedings of the Koli Calling International Conference on Computing Education Research*. 122–126.
[105] H. Chad Lane and Kurt VanLehn. 2005. Teaching the tacit knowledge of programming to novices with natural language tutoring. *Comput. Sci. Edu.* 15, 3 (2005), 183–201.
[106] Timotej Lazar and Ivan Bratko. 2014. Data-driven program synthesis for hint generation in programming tutors. In *Intelligent Tutoring Systems*. Springer, 306–311.
[107] Nguyen-Thinh Le. 2016. A classification of adaptive feedback in educational systems for programming. *Systems* 4, 2 (2016).
[108] Nguyen-Thinh Le and Wolfgang Menzel. 2006. Problem solving process oriented diagnosis in logic programming. In *Proceedings of the Conference on Computers in Education*. 63–70.
[109] Nguyen-Thinh Le, Wolfgang Menzel, and Niels Pinkwart. 2009. Evaluation of a constraint-based homework assistance system for logic programming. In *Proceedings of the Conference on Computers in Education*. 51–58.
[110] Nguyen-Thinh Le and Niels Pinkwart. 2011. Adding weights to constraints in intelligent tutoring systems: Does it improve the error diagnosis? In *Towards Ubiquitous Learning*. LNCS 6964. 233–247.
[111] Nguyen-Thinh Le and Niels Pinkwart. 2011. INCOM: A web-based homework coaching system for logic programming. In *Proceedings of the Conference on Cognition and Exploratory Learning in Digital Age*. 43–50.
[112] Nguyen-Thinh Le and Niels Pinkwart. 2014. Towards a classification for programming exercises. In *Proceedings of the Workshop on AI-supported Education for Computer Science*. 51–60.
[113] Nguyen-Thinh Le, Sven Strickroth, Sebastian Gross, and Niels Pinkwart. 2013. A review of AI-supported tutoring approaches for learning programming. In *Advanced Computational Methods for Knowledge Engineering*. Springer, 267–279.
[114] Chee-Kit Looi. 1991. Automatic debugging of prolog programs in a prolog intelligent tutoring system. *Instruct. Sci.* 20, 2–3 (1991), 215–263.
[115] Susan Lowes. 2007. *Online Teaching and Classroom Change: The Impact of Virtual High School on Its Teachers and Their Schools*. Technical Report. Columbia University, Institute for Learning Technologies.
[116] Cara MacNish. 2000. Java facilities for automating analysis, feedback and assessment of laboratory work. *Comput. Sci. Edu.* 10, 2 (2000), 147–163.
[117] Cara MacNish. 2002. Machine learning and visualisation techniques for inferring logical errors in student code submissions. In *Proceedings of the IEEE Conference on Advanced Learning Technologies*. 317–321.

[118] Tim A. Majchrzak and Claus A. Usener. 2013. Evaluating the synergies of integrating e-assessment and software testing. In *Information Systems Development*. Springer, New York, NY, 179–193.
[119] Amit Kumar Mandal, Chittaranjan Mandal, and Chris Reade. 2007. A system for automatic evaluation of programs for correctness and performance. In *Proceedings of the Conferences on Web Information Systems and Technologies 2005 and 2006*. 367–380.
[120] Fatima Z. Mansouri, Cleveland A. Gibbon, and Colin A. Higgins. 1998. PRAM: Prolog automatic marker. In *Innovation and Technology in Computer Science Education*. ACM, 166–170.
[121] Roozbeh Matloobi, Michael Blumenstein, and Steve Green. 2007. An enhanced generic automated marking environment: GAME-2. *IEEE Multidisc. Eng. Edu. Mag.* 2, 2 (2007), 55–60.
[122] Roozbeh Matloobi, Michael Blumenstein, and Steve Green. 2009. Extensions to generic automated marking environment: Game-2+. In *Proceedings of the Interactive Computer Aided Learning Conference*, Vol. 1. 1069–1076.
[123] Gordon McCalla, Richard Bunt, and Janelle Harms. 1986. The design of the SCENT automated advisor. *Comput. Intell.* 2, 1 (1986), 76–92.
[124] Gordon McCalla, Jim Greer, Bryce Barrie, and Paul Pospisil. 1992. Granularity hierarchies. *Comput. Math. Appl.* 23, 2–5 (1992), 363–375.
[125] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Proceedings of the Working Group Reports of Innovation and Technology in Computer Science Education*. 125–180.
[126] Jean McKendree, Bob Radlinski, and Michael E. Atwood. 1992. The grace tutor: A qualified success. In *Intelligent Tutoring Systems*. Springer, 677–684.
[127] Douglas C. Merrill, Brian J. Reiser, Michael Ranney, and J. Gregory Trafton. 1992. Effective tutoring techniques: A comparison of human tutors and intelligent tutoring systems. *J. Learn. Sci.* 2, 3 (1992), 277–305.
[128] Antonija Mitrovic, Kenneth Koedinger, and Brent Martin. 2003. A comparative analysis of cognitive tutoring and constraint-based modeling. In *User Modeling*. Springer, 313–322.
[129] Joseph Moghadam, Rohan Roy Choudhury, HeZheng Yin, and Armando Fox. 2015. AutoStyle: Toward coding style feedback at scale. In *Proceedings of the ACM Conference on Learning at Scale*. 261–266.
[130] William R. Murray. 1987. Automatic program debugging for intelligent tutoring systems. *Comput. Intell.* 3, 1 (1987), 1–16.
[131] Susanne Narciss. 2008. Feedback strategies for interactive learning tasks. *Handbook of Research on Educational Communications and Technology*. Routledge, 125–144.
[132] Peter Naur. 1964. Automatic grading of student's ALGOL programming. *BIT Numer. Math.* 4, 3 (1964), 177–188.
[133] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. 2014. Codewebs: Scalable homework search for massive open online programming courses. In *Proceedings of the Conference on World wide web*. 491–502.
[134] Elizabeth Odekirk-Hash and Joseph L. Zachary. 2001. Automated feedback on programs means students need less help from teachers. *ACM SIGCSE Bull.* 33, 1 (2001), 55–59.
[135] Claudia Ott, Anthony Robins, and Kerry Shephard. 2016. Translating principles of effective feedback for students into the CS1 context. *ACM Trans. Comput. Edu.* 16, 1, Article 1 (2016), 27 pages.
[136] Martin Pärtel, Matti Luukkainen, Arto Vihavainen, and Thomas Vikberg. 2013. Test my code. *Int. J. Technol. Enhanced Learn.* 5, 3–4 (2013), 271–283.
[137] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bull.* 39, 4 (2007), 204–223.
[138] Daniel Perelman, Judith Bishop, Sumit Gulwani, and Dan Grossman. 2015. *Automated Feedback and Recognition through Data Mining in Code Hunt, MSR-TR-2015-57*. Technical Report. Microsoft Research.
[139] Daniel Perelman, Sumit Gulwani, and Dan Grossman. 2014. Test-driven synthesis for automated feedback for introductory computer science assignments. In *Proceedings of the Workshop on Data Mining for Educational Assessment and Feedback*.
[140] Raymond Pettit and James Prather. 2017. Automated assessment tools: Too many cooks, not enough collaboration. *J. Comput. Sci. Colleges* 32, 4 (2017), 113–121.
[141] Christoph Peylo, Tobias Thelen, Claus Rollinger, and Helmar Gust. 2000. A web-based intelligent educational system for PROLOG. In *Proceedings of the Workshop on Adaptive and Intelligent Web-Based Education Systems, ITS*. 70–80.
[142] Nelishia Pillay. 2003. Developing intelligent programming tutors for novice programmers. *ACM SIGCSE Bull.* 35, 2 (2003), 78–82.
[143] Yusuf Pisan, Debbie Richards, Anthony Sloane, Helena Koncek, and Simon Mitchell. 2002. Submit! A web-based system for automatic program critiquing. In *Proceedings of the Australasian Conference on Computing Education*. 59–68.

[144] Ivan Pribela, Mirjana Ivanović, and Zoran Budimac. 2011. System for testing different kinds of students' programming assignments. In *Proceedings of the Conference on Information Technology*.
[145] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Trans. Comput. Edu.* 18, 1 (2017), 1.
[146] Rob Radlinski and Jean McKendree. 1992. Grace meets the real world: Tutoring COBOL as a second language. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 343–350.
[147] Khirulnizam Abd Rahman and Md Jan Nordin. 2007. A review on the static analysis approach in the automated programming assessment systems. In *Proceedings of the National Conference on Programming*.
[148] Haider A. Ramadhan, Fadi Deek, and Khalil Shihab. 2001. Incorporating software visualization in the design of intelligent diagnosis systems for user programming. *Artific. Intell. Rev.* 16, 1 (2001), 61–84.
[149] Kelly Rivers and Kenneth Koedinger. 2015. Data-driven hint generation in vast solution spaces: A self-improving python programming tutor. *Int. J. Artific. Intell. Edu.* 27, 1 (2015), 37–64.
[150] Juan Carlos Rodríguez-del Pino, Enrique Rubio-Royo, and Zenón Hernández-Figueroa. 2012. A virtual programming lab for moodle with automatic assessment and anti-plagiarism features. In *Proceedings of the Conference on e-Learning, e-Business, Entreprise Information Systems, & e-Government*.
[151] Rohaida Romli, Shahida Sulaiman, and Kamal Zuhairi Zamli. 2010. Automatic programming assessment and test data generation. In *Proceedings of the International Symposium in Information Technology*. 1186–1192.
[152] Tammy Rosenthal, Patrick Suppes, and Nava Ben-Zvi. 2002. Automated evaluation methods with attention to individual differences—A study of a computer-based course in C. In *Proceedings of the Frontiers in Education Conference*, Vol. 1. 7–12.
[153] Gregory R. Ruth. 1976. Intelligent program analysis. *Artific. Intell.* 7 (1976), 65–85.
[154] Warren Sack. 1992. Knowledge base compilation and the language design game. In *Intell. Tutor. Syst.* 225–233.
[155] Warren Sack and Elliot Soloway. 1992. From PROUST to CHIRON: ITS design as iterative engineering; intermediate results are important! *Comput.-Assist. Instruct. Intell. Tutor. Syst.: Shared Goals Complement. Approaches* (1992), 239–274.
[156] Riku Saikkonen, Lauri Malmi, and Ari Korhonen. 2001. Fully automatic assessment of programming exercises. In *ACM SIGCSE Bulletin*, Vol. 33. 133–136.
[157] Joseph A. Sant. 2009. Mailing it in: Email-centric automated assessment. *ACM SIGCSE Bull.* 41, 3 (2009), 308–312.
[158] Steven C. Shaffer. 2005. Ludwig: An online programming tutoring and assessment system. *ACM SIGCSE Bull.* 37, 2 (2005), 56–60.
[159] Goran Shimic and Aleksandar Jevremovic. 2012. Problem-based learning in formal and informal learning environments. *Interact. Learn. Environ.* 20, 4 (2012), 351–367.
[160] Valerie J. Shute. 2008. Focus on formative feedback. *Rev. Edu. Res.* 78, 1 (2008), 153–189.
[161] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. *ACM SIGPLAN Notices* 48, 6 (2013), 15–26.
[162] Matthew Z. Smith and Joseph J. Ekstrom. 2004. String of perls : Using perl to teach perl. In *Proceedings of the ASEE Annual Conference and Exposition*.
[163] Elliot Soloway, Eric Rubin, Beverly Woolf, Jeffrey Bonar, and W. Lewis Johnson. 1983. Meno-II: An AI-based programming tutor. *J. Comput.-Based Instruct.* 10, 1 (1983).
[164] J. S. Song, S. H. Hahn, K. Y. Tak, and J. H. Kim. 1997. An intelligent tutoring system for introductory C language course. *Comput. Edu.* 28, 2 (1997), 93–102.
[165] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A review of generic program visualization systems for introductory programming education. *ACM Trans. Comput. Edu.* 13, 4 (2013), 1–64.
[166] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K. Hollingsworth, and Nelson Padua-Perez. 2006. Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses. *ACM SIGCSE Bull.* 38, 3 (2006), 13–17.
[167] Michael Striewe, Moritz Balz, and Michael Goedicke. 2009. A flexible and modular software architecture for computer aided assessments and automated marking. *Proceedings of the Conference on Computer Supported Education* 2 (2009), 54–61.
[168] Michael Striewe and Michael Goedicke. 2011. Using run time traces in automated programming tutoring. In *Innovation and Technology in Computer Science Education*. ACM, 303–307.
[169] Michael Striewe and Michael Goedicke. 2014. A review of static analysis approaches for programming exercises. In *Computer Assisted Assessment. Research into E-Assessment*. Springer, 100–113.
[170] Ryo Suzuki, Gustavo Soares, Elena Glassman, Andrew Head, Loris D'Antoni, and Björn Hartmann. 2017. Exploring the design space of automatically synthesized hints for introductory programming assignments. In *Proceedings of the SIGCHI Conference Extended Abstracts on Human Factors in Computing Systems*. 2951–2958.

[171] Edward Sykes. 2005. Qualitative evaluation of the java intelligent tutoring system. *J. System., Cybernet. Info.* 3, 5 (2005), 49–60.
[172] Edward Sykes. 2010. Design, development and evaluation of the java intelligent tutoring system. *Technol., Instruct., Cogn. Learn.* 8, 1 (2010), 25–65.
[173] Gareth Thorburn and Glenn Rowe. 1997. PASS: An automated system for program assessment. *Comput. Edu.* 29, 4 (1997), 195–206.
[174] Nikolai Tillmann, Jonathan de Halleux, Tao Xie, Sumit Gulwani, and Judith Bishop. 2013. Teaching and learning programming and software engineering via interactive gaming. In *Proceedings of the Conference on Software Engineering*. 1117–1126.
[175] Nghi Truong, Peter Bancroft, and Paul Roe. 2005. Learning to program through the web. *ACM SIGCSE Bull.* 37, 3 (2005), 9–13.
[176] Nghi Truong, Paul Roe, and Peter Bancroft. 2004. Static analysis of students' Java programs. In *Proceedings of the Australasian Conference on Computing Education*, Vol. 30. 317–325.
[177] Haruki Ueno. 2000. A generalized knowledge-based approach to comprehend pascal and C programs. *IEICE Trans. Info. Syst.* 83, 4 (2000), 591–598.
[178] Miguel Ulloa. 1980. Teaching and learning computer programming: A survey of student problems, teaching methods, and automated instructional tools. *ACM SIGCSE Bull.* 12, 2 (1980), 48–64.
[179] Alexandria Katarina Vail and Kristy Elizabeth Boyer. 2014. Identifying effective moves in tutoring: On the refinement of dialogue act annotation schemes. In *Intelligent Tutoring Systems*. Springer, 199–209.
[180] Jeroen Van Merriënboer and Marcel De Croock. 1992. Strategies for computer-based programming instruction: Program completion vs. program generation. *J. Edu. Comput. Res.* 8, 3 (1992), 365–94.
[181] Kurt VanLehn. 2006. The behavior of tutoring systems. *Int. J. Artific. Intell. Edu.* 16, 3 (2006), 227–265.
[182] Philip Vanneste, Koen Bertels, and Bart Decker de. 1996. The use of reverse engineering to analyse student computer programs. *Instruct. Sci.* 24 (1996), 197–221.
[183] Anne Venables and Liz Haywood. 2003. Programming students NEED instant feedback! In *Proceedings of the Australasian Conference on Computing Education*, Vol. 20. 267–272.
[184] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. 2013. Scaffolding students' learning using test my code. In *Innovation and Technology in Computer Science Education*. ACM, 117–122.
[185] Aurora Vizcaíno. 2005. A simulated student can improve collaborative learning. *Int. J. Artific. Intell. Edu.* 15 (2005), 3–40.
[186] Aurora Vizcaíno, Juan Contreras, Jesús Favela, and Manuel Prieto. 2000. An adaptive, collaborative environment to develop good habits in programming. In *Intelligent Tutoring Systems*. LNCS 1839. 262–271.
[187] Milena Vujošević-Janičić, Mladen Nikolić, Dušan Tošić, and V. Kuncak. 2013. Software verification and graph similarity for automated evaluation of students' assignments. *Info. Softw. Technol.* 55, 6 (2013), 1004–1016.
[188] Tiantian Wang, Xiaohong Su, Peijun Ma, Yuying Wang, and Kuanquan Wang. 2011. Ability-training-oriented automated assessment in introductory programming course. *Comput. Edu.* 56, 1 (2011), 220–226.
[189] Gerhard Weber. 1996. Episodic learner modeling. *Cogn. Sci.* 20, 2 (1996), 195–236.
[190] Gerhard Weber and Peter Brusilovsky. 2001. ELM-ART: An adaptive versatile system for web-based instruction. *Int. J. Artific. Intell. Edu.* 12 (2001), 351–384.
[191] Gerhard Weber and Marcus Specht. 1997. User modeling and adaptive navigation support in WWW-based tutoring systems. In *Proceedings of the Conference on User Modeling*. 289–300.
[192] Dinesha Weragama. 2013. *Intelligent Tutoring System for Learning PHP*. Ph.D. Dissertation. Queensland University of Technology.
[193] Dinesha Weragama and Jim Reye. 2014. Analysing student programs in the PHP intelligent tutoring system. *Int. J. Artific. Intell. Edu.* 24, 2 (2014), 162–188.
[194] Weimin Wu, Guangqiang Li, Yinai Sun, Jing Wang, and Tianwu Lai. 2007. AnalyseC: A framework for assessing students' programs at structural and semantic level. In *Proceedings of the Conference on Control and Automation*. 742–747.
[195] Songwen Xu and Yam San Chee. 2003. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Trans. Softw. Eng.* 29, 4 (2003), 360–384.
[196] Cheng Yongqing, Hu Qing, and Yang Jingyu. 1988. An expert system for education: IPTS. In *Proceedings of the Conference on Systems, Man, and Cybernetics*. 930–933.

Received January 2018; revised April 2018; accepted May 2018