Towards a Systematic Review of Automated Feedback Generation for Programming Exercises – Extended Version

Hieke Keuning

Johan Jeuring

Bastiaan Heeren

Technical Report UU-CS-2016-001
March 2016

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Towards a Systematic Review of Automated Feedback Generation for Programming Exercises – Extended Version*

Hieke Keuning
Open University of the Netherlands and Windesheim University of Applied Sciences
hw.keuning@windesheim.nl

Johan Jeuring
Utrecht University and Open University of the Netherlands
j.t.jeuring@uu.nl

Bastiaan Heeren
Open University of the Netherlands
bastiaan.heeren@ou.nl

ABSTRACT

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 are performing 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 the first iteration of our search in which we coded 69 tools. We have found that tools do not often give feedback on fixing problems and taking a next step, and that teachers cannot easily adapt tools to their own needs.

Keywords

systematic literature review, automated feedback, programming tools, learning programming

1. INTRODUCTION

Tools that support students in learning programming have been developed since the 1960s [44]. Such tools provide a simplified development environment, use visualisation or animation to give better insight in running a program, guide students towards a correct program by means of hints and feedback messages, or automatically grade the solutions of students [79].

Two important reasons to develop tools that support learning programming are:

- learning programming is hard [102], and students need help to make progress [26];
- programming courses are taken by many thousands of students all over the world [16], and helping students individually with their problems requires a huge time investment of teachers [110].

Feedback is an important factor in learning [124, 64]. Boud and Molloy define feedback as 'the process whereby learners obtain information about their work in order to appreciate the similarities and differences between the appropriate standards for any given work, and the qualities of the work itself, in order to generate improved work' [22]. 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' [124]. 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 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 paper 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 [103].

Given the role of feedback in learning, 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 are performing 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' [81]. An SLR results in a thorough and fair examination of a particular topic. A research plan is designed in advance, and the execution of this plan is documented in detail, allowing insight into the rigorousness of the research.

This paper reports on the results found in the first iteration of our search for relevant papers. We searched for related overviews on tools for learning programming and executed the first step of 'backward snowballing' by selecting relevant references from the papers we found. In the future, we intend to repeatedly search the papers we have found so far for relevant references, until we do not find new papers. Furthermore, because our questions are related to computer science and education, we plan to search a computer science database (ACM Digital Library), an educational database (ERIC), and a general scientific database (Scopus). Our

*This is the extended version of our paper [80].

search until now has resulted in a set of 102 papers, describing 69 different tools. Although not yet complete, this collection is representative and large enough to report some findings.

We have classified these tools by means of Narciss' [107] 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. For example, in the category 'knowledge about mistakes' we have introduced the subcategories: 'compiler errors', 'solution errors', and 'performance issues'. Narciss' categories largely overlap with the categories used to describe the actions of human tutors when they help students learning programming [104, 138].

We not only classify the kind of feedback given by the various tools that support learning programming, but also determine *how* these tools generate this feedback. What are the underlying techniques used to generate feedback? An answer to this question allows us to relate techniques to feedback types and identify possibilities and limitations. Here we will build upon an earlier review in which static analysis approaches for supporting programming exercises are categorised and compared [128].

Besides looking at feedback categories (the output of a tool) and the techniques used to generate feedback (what happens inside a tool), we also look at the input of a tool. The input of a tool supporting learning programming may take the form of model programs, test cases, templates, feedback messages, etc., and determines to a large extent the adaptability of the tool. Adaptability of the content of a learning tool is important [20, 37, 96].

Finally, we collect information about the effectiveness of the feedback generated by the various tools. The effectiveness of a tool depends on many factors (for example, Narciss [107] identifies 'presentation' and 'individual and situational conditions') and tools have been evaluated by a large variety of methods. Gross and Powers [61] distinguish 'anecdotal', 'analytical' and 'empirical' as the main categories. Because of this large variety, it is difficult to compare tools.

This review makes the following contributions:

- We analyse what kind of feedback is used in tools that support a student in learning programming. Although quite a few reviews analyse such tools, none of them specifically looks at the feedback provided by these tools.
- We relate the feedback content to its technology, and the adaptability of the tool and the feedback.

The most remarkable results that have emerged from this research so far are:

- Very few tools that support exercises that can be solved by multiple (variants of) strategies give feedback with 'knowledge on how to proceed'. According to Boud and Molloy's definition, these tools lack the means to really help a student.
- In general, the feedback that tools generate is not that diverse, and mainly focused on identifying mistakes.
- Teachers cannot easily adapt tools to their own needs, except for test-based AA systems.

This paper is organised as follows. Section 2 discusses related reviews of tools for learning programming. Section 3 gives our research questions and research method. Section 4 describes the labelling and shows the results. Section 5 discusses the results and Section 6 concludes the paper and describes future work.

2. RELATED WORK

We have found almost twenty reviews of tools for learning programming, mostly on learning environments for programming (Section 2.1) or automated assessment (AA) tools (Section 2.2). 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 identify the main research questions of the related review papers, the scope of the selected tools and the method of data collection. Section 2.3 discusses how our paper differs from the reviews discussed in this section, and draws some conclusions.

2.1 Reviews of learning tools

Ulloa [137] describes several tools and other methods for teaching programming, to better understand the difficulties of novice programmers. He identifies two types of automated tools: interactive tools that automate teaching, and noninteractive tools, or student-oriented compilers. The review describes the behaviour of these tools, but does not elaborate on the techniques used.

Deek and McHugh [43] classify tools for learning programming as programming environments, debugging aids, intelligent tutoring systems, or intelligent programming environments. They illustrate each of these categories with multiple examples, and analyse their benefits and limitations. Deek et al. [42] continue this work by investigating *web-based*, interactive instructional systems for programming. Their goal is to help educators to select systems that fit their needs. They distinguish three categories of instructional systems: drill and practice systems, tutorial systems and simulation systems. Each category is illustrated with several examples, and is analysed with respect to nine instructional events identified by Gagné et al. [53], one of which is 'provide feedback about performance'.

Pillay [114] focuses on Intelligent Tutoring Systems (ITSs) for programming. She discusses the development and architecture of ITSs for programming, and briefly describes a number of ITSs.

Guzdial [62] analyses programming environments for novices. He distinguishes three categories of environments: Logo and its descendants, rule-based programming, and extended programming environments for traditional programming languages. Each of these categories is illustrated with several examples, and various trends are identified.

Kelleher and Pausch [79] present a taxonomy of programming languages and environments for novices. The taxonomy consists of two large groups: teaching systems and empowering systems. Each of these groups of systems is divided into subgroups multiple times. Every group in the taxonomy is illustrated with several examples, and the attributes (such as supported programming style, supported constructs and code representation) of the systems that are cited most are identified.

Gómez-Albarrán [58] presents a classification of tools for learning programming. She distinguishes four categories of tools: tools with a reduced development environment,

example-based environments, tools based on visualization and animation, and simulation environments. For each category she selects a number of widely used tools, and describes their main features. Finally, she identifies some challenges for future work.

Pears et al. [113] give a survey of research on teaching introductory programming. The results described in this paper are based on papers selected by an ITiCSE working group, together with paper suggestions by ten academics outside the working group. One of the subareas they consider is tools. Their purpose is not to survey tools research, but instead to give an outline of the field of teaching introductory programming, and to point out sub-areas that may be particularly relevant to teachers of introductory programming courses. They distinguish and describe four categories of tools: visualization tools, automated assessment tools, programming environments for novices, and other tools, including intelligent tutoring systems for programming.

Le et al. [91] review AI-supported tutoring approaches for programming. These approaches are based on examples, simulations, collaboration, dialogues, program analysis, or feedback (which is used in the other tutoring approaches as well). The authors identify examples of tools for each of the categories, and describe the AI-techniques used.

Nesbit et al. [109] present work in progress on a systematic review of Intelligent Tutoring Systems in computer science and software engineering education. The authors define an ITS as any system 'that performs teaching or tutoring functions (..) and adapts or personalizes those functions by modelling students' cognitive, motivational or emotional states', thus emphasizing student modelling. The aim of the review is to examine the research field and identify its features (subject areas, instructional functions and strategies, types of student modelling, et cetera). The work in progress paper describes the research method and search process in detail and reports on the first results of four basic variables (publication date, publication type, educational level and subject domain). The automatic generation of hints is one of the popular themes that the paper identifies.

Although not a review of learning tools, we also include the work of Le and Pinkwart on classifying programming exercises supported in learning environments [90]. 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 quiz-like 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 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. The authors describe several tools they examined for the development of this classification.

2.2 Reviews of assessment tools

Ala-Mutka [2] describes the different aspects of programs that are assessed in automated assessment tools, ranging from aspects that are statically analysed, such as style, design, and metrics, to aspects that are mainly dynamically analysed, such as input-output behavior and efficiency. These aspects are illustrated with actual assessment tools. The goal of this survey is to promote automated assessment and to provide readers with starting points for further research. Ihantola et al. [69] follow up on the review from Ala-Mutka [2]. They review research on automated assessment in the period 2006–2010 using a systematic literature review. They only include papers on tools that provide summative, numerical feedback. The research questions aim to identify the features of assessment tools from the period, and ideas for future research. The authors distinguish two main categories: automatic assessment systems for programming competitions and automatic assessment systems for (introductory) programming education. On their turn, Caiza and Ramiro [25] follow up on the review of Ihantola et al., describing a collection of mature as well as recent assessment systems, particularly looking at the evolution of these systems since the review of Ihantola et al. [69].

Douce et al. [44] focus on test-based assessment, aiming to inform and guide future developments. They also discuss pedagogic issues. Tools are arranged as first (early), second (tool-oriented) or third (web-oriented) generation. The authors briefly describe their search strategy.

Rahman and Nordin [115] give an overview of static analysis methods and identify their advantages and disadvantages.

Liang et al. [94] describe recent developments in the field of automated assessment for programming, and refer to several tools as an example.

Romli et al. [119] provide a review of automated assessment systems, focusing on approaches for test data generation. They describe the characteristics and features (assessment method, quality factors) of a substantial number of systems, and identify trends.

Striewe and Goedicke [128] specifically focus on the didactic benefits of static analysis techniques used in assessment tools. The paper has quite a narrow scope: it only considers tools that run on a server, targeted at object-oriented Java programming, and exclude metrics-based tools. The paper does include tools that help a student finishing an incomplete solution, and discusses the configurability of the assessment tools.

2.3 Conclusion

Most review papers describe the features and characteristics of a number of tools, identify challenges, and direct future research. Except for the review by Ihantola et al. [69] and the review in progress by Nesbit et al. [109], 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. 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 learning tools for programming. Furthermore, we employ a more systematic approach than almost all of the above papers: tools are selected in a systematic way, following strict criteria, and are coded 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.

3.1 Research questions

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

RQ1. What is the nature of the feedback that is generated? We classify the content of feedback messages using Narciss' [107] categories of feedback.

RQ2. Which techniques are used to generate the feedback?

RQ3. How can the tool be adapted by teachers, to create exercises and to influence the feedback?

RQ4. 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 [79], 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.

The rationale of our functionality criteria is that the ability to develop a program that solves a particular problem is an important learning objective for learning programming [77]. 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 [41]. We do not include visualisation tools for programming because they have been surveyed extensively by Sorva et al. [125] in the recent past.

We select papers and tools that satisfy all inclusion criteria and none of the exclusion criteria. We have included some theses, because either they have been cited often, or their contribution is documented in another published paper. Since no review addressing our research questions has been conducted before, and we aim for a complete overview of the field, we do not exclude papers based on publication date.

3.3 Search process

The starting point of our search for papers was the collection of 17 review papers given 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. We had to exclude a small number of papers that 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.

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.

Starting with an initial selection of 197 papers, we ended up with a total of 102 papers describing 69 different tools.

4. RESULTS

To systematically encode the information in the papers, we use a labelling based on the answers to the research questions we expected to get, refined by encoding a small set of randomly selected papers. One of the authors encoded the complete set of papers. Whenever there were questions about the coding of a paper, another author checked. In total, 28% of the codings were handled by two authors. A third author joined the general discussions about the coding. When necessary, we made adjustments to the labelling. With this process we have coded the information to the best of our ability, although we cannot ensure that our coding does not contain any mistake.

The results we have found so far in our SLR are shown in Table 2 and Table 3.[1] This section discusses the preliminary results, and describes the labelling. The next subsections focus on the four research questions, but first we discuss the general properties of the tools we investigated. We refer to the tools in the table by their name in a Small caps font, or the first author and year of the most recent paper (Author00) on the tool we have used.

Programming language

Tools offer either exercises for a specific programming language, a set of programming languages within a particular paradigm, or multiple languages within multiple paradigms. From the 69 tools we found, a majority of 70% supports programming in imperative languages, including object-oriented languages. Tools developed in the 21st century often support languages such as Java, C and C++, whereas older tools provide exercises in ALGOL (Naur64 [108]), FORTRAN (LAURA [1]) and Ada (ASSYST [71]). Of all tools, 7% support a functional programming language such as Lisp (The LISP tutor [7]), Scheme (Scheme-robo [120]) or Haskell (Ask-Elle [74]). All tools for logic programming (6%) offer exercises in Prolog. Three recent tools (4%) focus on web scripting languages such as PHP and JavaScript. The remaining tools support multiple languages of different types and paradigms, and are often test-based AA systems.

Programming exercises often require the student to write a few lines of code or a single function, meaning that 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.

[1]The table with the results can also be accessed online at www.open.ou.nl/xhk/review.

Table 1: Criteria for the inclusion/exclusion of papers

	Include	Exclude
General	Scientific publications (journal papers and conference papers) in English. Master 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 Lee and Pinkwart [90] (see Section 2.1). The tool provides automated, textual feedback on (partial) solutions, targeted at the student.	Tools that only produce a grade.
Domain	Tools that support a high-level, general purpose, textual programming language, including pseudo-code.	Visual programming tools (programming with blocks, flowcharts). Tools that only teach a particular aspect of programming, such as recursion or multi-threading.

Exercise type

We record the highest exercise class a tool supports. We have found that 20% of the tools support exercises of class 2, and 80% exercises of class 3.

4.1 RQ1 Feedback type

Narciss [107] 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, metacognitive and motivational) and *presentation* (timing, number of tries, adaptability, 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 as well as summative).

Narciss first gives *simple* feedback components: 'Knowledge of performance for a set of tasks', 'Knowledge of result/response' and 'Knowledge of the correct results'. These types of feedback are not intended to 'generate improved work', a requirement in the feedback definition by Boud and Molloy. Because we focus on formative feedback on a single exercise, we do not identify these types in our coding.

Knowledge of performance for a set of tasks (KP)

KP is summative feedback on the achieved performance level after doing multiple tasks. Examples from Narciss are '15 of 20 correct' and '85% correct'.

Knowledge of result/response (KR)

This type of feedback communicates whether a solution is correct or incorrect. Correctness may have a different meaning in different tools. We have found the following meanings of 'correctness of a solution':

- it passes all tests,
- it is equal to a model program,
- it satisfies one or more constraints,
- a combination of the above.

Some tools are able to assess partial solutions that are correct but not yet final.

Knowledge of the correct results (KCR)

KCR is a description or indication of a correct solution.

The next five types are *elaborated* feedback components. Each type addresses an element of the instructional context. We provide several examples to illustrate how the different types are instantiated in learning tools for programming. We have also identified a few examples of feedback messages that do not fit into our classification, such as motivating feedback.

Knowledge about task constraints (KTC)

KTC deals with feedback on task rules, task constraints, and task requirements. We define two labels for this type.

- 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. A concrete example can be found in the INCOM system [89]. 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' [87]. Another example can be found in the BASIC Instructional Program (BIP) [10]. Some exercises in BIP require the use of a specific language construct. If this construct is missing from the student solution, the stu-

Table 2: Class 2 tools

			RQ1 Feedback type											RQ2 Technique								RQ3 Adaptability						RQ4 Evaluation				
			KTC		KC		KM					KH		KMC	MT	CBM	AT	BSA	PT	IBD	EX	Other	ST	MS	TD	ED	SM	Other	ANC	ANL	EM-LO	EM-SU	EM-TA
Name, reference	Language	Ex. class	TR	TPR	EXP	EXA	TF	CE	SE	SI	PI	EC	TPS																				
ACT Programming Tutor (APT) [35, 33, 34]	Multi	C2		●			○		●				●		●		●													●	●		
Bridge [21]	Imp/OO	C2							●			●	●		●															●			
(Chang00) [27]	Imp/OO	C2							●											●			●	●						●	●		
DISCOVER [116]	Imp/OO	C2						●	●			◐	●		●									●				●		●	●		
ELP [134, 135]	Imp/OO	C2					●		●			◑					●	●	●				●	●	●			●				●	
HabiPro [141, 142]	Imp/OO	C2				●						●		●								●	●	●			●	●			●	●	
INTELLITUTOR (II) [136]	Imp/OO	C2							●			◐						●	●														●
InSTEP [112]	Imp/OO	C2					●	●	●			◐					●	●					●		●						●		
JITS [131, 130]	Imp/OO	C2					○	○				●					●	●	●				●		●					●	●	●	
LAURA [1]	Imp/OO	C2							●										●					●									●
PASS [133]	Imp/OO	C2					●		●			◐					●		●	●				●									●
ProPL [85]	Imp/OO	C2	●						●			●	●									●			●						●		
RoboProf [39, 38]	Imp/OO	C2					●										●							●	●						●		
The LISP tutor [36, 7]	Fun	C2				●			●			●	●		●															●	●		

RQ1		RQ2		RQ3		RQ4	
KTC	Knowledge about task constraints	MT	Model tracing	ST	Solution templates	ANC	Anecdotal assessment
TR	Hints on task requirements	CBM	Constraint-based modelling	MS	Model solutions	ANL	Analytical assessment
TPR	Hints on task-processing rules	AT	Automated testing	TD	Test data	EM-LO	Empirical - Learning outcome evaluations
KC	Knowledge about concepts	SA	Basic static code analysis	ED	Error data	EM-SU	Empirical - Surveys
EXP	Explanations on subject matter	PT	Program transformations	SM	Student model	EM-TA	Empirical - Technical analysis
EXA	Examples illustrating concepts	IBD	Intension-based diagnosis				
KM	Knowledge about mistakes	EX	External tools				
	(○ basic or ● detailed)						
TF	Test failures						
CE	Compiler errors						
SE	Solution errors						
SI	Style issues						
PI	Performance issues						
KH	Knowledge about how to proceed						
	(◐ hint, ◑ solution or ● both)						
EC	Bug-related hints for error correction						
TPS	Task-processing steps						
KMC	Knowledge about meta-cognition						

Table 3: Class 3 tools

			RQ1 Feedback type											RQ2 Technique								RQ3 Adaptability						RQ4 Evaluation				
			KTC		KC		KM					KH		KMC																			
Name, reference	Language	Ex. class	TR	TPR	EXP	EXA	TF	CE	SE	SI	PI	EC	TPS		MT	CBM	AT	BSA	PT	IBD	EX	Other	ST	MS	TD	ED	SM	Other	ANC	ANL	EM-LO	EM-SU	EM-TA
ADAPT [54]	Log	C3		●	●	●			●				●						●	●				●								●
APOGEE [52]	Web	C3					●										●						●		●							
APROPOS2 [95]	Log	C3							●			●					●	●	●	●												●
ASAP [45, 46]	Imp/OO	C3					●										●								●						●	
ASSYST [73, 71]	Imp/OO	C3					○			●	●						●	●						●	●				●				
AnalyseC [148]	Imp/OO	C3							●			◑						●	●					●									●
Ask-Elle [74, 56, 55]	Fun	C3			●		●	●					●		●		●		●			●		●	●			●				●	●
AutoGrader [65]	Imp/OO	C3					●			●							●				●		●	●	●			●	●				
AutoLEP [144]	Imp/OO	C3					●	●	●								●	●	●					●	●						●	●	●
Automatic Marker for Sakai [129]	Imp/OO	C3					●	●									●								●							●	
BIP [10, 11, 12]	Imp/OO	C3	●	●	●		●	●				◐					●	●						●	●			●			●		
BOSS [78]	Multi	C3					○										●						●		●					●		●	
(Bettini04) [17]	Imp/OO	C3					●	●									●						●	●	●						●	●	
Ceilidh [15, 13, 14]	Multi	C3					●	●	○	○	○						●	●			●		●	●	●			●				●	
(Chen04) [29]	Imp/OO	C3					●										●							●	●	●			●				
(Coffman10) [32]	Web	C3					●										●						●		●								
CourseMarker/CourseMaster [66, 67, 51]	Multi	C3			●		●	●	●	●	●						●	●			●		●	●	●			●			●	●	
ELM-PE/ELM-ART (II) [147, 145, 23, 146]	Fun	C3			●	●	●		●			●	●				●		●	●							●			●	●	●	●
EduComponents [5]	Multi	C3					●	●									●							●	●							●	
(Fischer06) [50]	Imp/OO	C3	●				●	○		○							●							●	●						●		
GAME (2, 2+) [100, 18, 19, 99]	Multi	C3					○	●	○	○							●	●						●	●								●
HOGG [106]	Imp/OO	C3					●										●						●	●	●				●				
(Harris04) [63]	Imp/OO	C3					●	●									●								●							●	
(Hong04) [68]	Log	C3							●			◐	◐						●	●				●									●
INCOM [89, 88, 86]	Log	C3	●	●					●			◐				●			●									●			●	●	●
ITEM/IP [24]	Imp/OO	C3		●			○										●							●	●						●		
(Isaacson89) [70]	Imp/OO	C3					○	○									●								●								
JACK [127, 57, 83, 82]	Imp/OO	C3	●				●	●	●								●	●	●				●		●	●		●			●	●	●
(Jackson00) [72]	Imp/OO	C3					●			●							●	●						●	●				●				
Kassandra [143]	Imp/OO	C3					●										●							●	●							●	
Ludwig [123]	Imp/OO	C3					●	●		●							●	●						●	●			●				●	
MarmoSet [126]	Multi	C3					●		●								●				●				●							●	
Mooshak [93, 92]	Imp/OO	C3					○	●									●								●				●				
(Naur64) [108]	Imp/OO	C3					●				○						●								●				●				
Online Judge [28]	Imp/OO	C3					○	○			○						●								●				●				
PASS [30, 31]	Imp/OO	C3					●										●								●							●	
PATTIE [40]	Imp/OO	C3		●	●				●				●		●							●					●		●				
PROUST [75]	Imp/OO	C3							●			◐							●	●													●
Praktomat [150]	Imp/OO	C3					●	●									●								●							●	
Quiver [49]	Imp/OO	C3					●	●									●						●	●	●				●				
RoboLIFT [3]	Imp/OO	C3					●										●						●		●						●		
SAC [9]	Imp/OO	C3					○	●									●								●				●				
SIPLeS-II [149]	Imp/OO	C3							●			◑						●	●					●				●					●
(Sant09) [121]	Imp/OO	C3					●			●							●				●				●				●				
Scheme-robo [120]	Fun	C3					●		○		○						●	●						●	●							●	
(Sztipanovits08) [132]	Web	C3					○										●								●								
TRY [117]	Multi	C3					●										●								●								
Virtual Programming Lab [118]	Multi	C3					●			●							●				●				●							●	
Web-CAT [48, 47]	Multi	C3					●		●	●							●				●			●	●			●			●	●	
WebToTeach [8]	Imp/OO	C3		●			●										●								●				●				
WebWork-JAG [60, 59]	Imp/OO	C3	●				●										●								●							●	
autograder [111]	Imp/OO	C3					●	●									●							●	●							●	
datlab [98, 97]	Imp/OO	C3					●		●								●					●		●								●	●
submit [140]	Imp/OO	C3					●	●		●							●	●							●							●	
xLx [122]	Imp/OO	C3					●	●									●							●	●								

dent will see the following message:

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 the 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 automated assessment tool by Fischer06 [50] provides the following feedback when a student uses a prohibited method:

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

– Hints on task-processing rules (TPR). These hints provide general information on how to approach the exercise and do not consider the current work of a student. The automated tutor in ADAPT gives some general information on how to solve a particular exercise [54]:

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 remove the first element from the input list and recursively invoke the main predicate with the tail of the list and then add the value of the head to the sum of the tail of the list.

Which component do you want to attempt first: 1. base case 2. recursive step 3. need further assistance?

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 define two labels for KC.

– Explanations on subject matter (EXP), generated while a student is working on an exercise. The Ask-Elle programming tutor [74] refers to relevant internet sources when a student encounters certain language constructs.

– Examples illustrating concepts (EXA). The LISP tutor [7, 36] uses examples in its tutoring dialogue. After a student has made a mistake, the tutor might respond with:

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 [146] 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.

Knowledge about mistakes (KM)

KM feedback messages have a type and a level of detail. A 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. Mooshak [92] is an automatic judge for programming contests that provides basic feedback on test results. The system attaches one of seven classification labels to a solution, such as 'wrong answer', 'presentation error' (correct output but formatting not as expected) or 'accepted'. Another programming contest judge, Online Judge [28], 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'). Coffman10 [32], an AA tool for web programming, provides detailed feedback on test results. The screenshot in Figure 1 shows the informative name of the test cases, a colour indicating their success and the reason why a particular test case failed. We found this type of feedback, which resembles the output of professional testing tools, in many AA tools.

Name	Started	Duration	Exception
checkName(1 , John Doe)	16:26:07	966 ms	java.lang.AssertionError:
checkLogin()	16:24:50	2s	
checkInvalidLogin()	16:25:00	2s	

Figure 1: Detailed feedback on test cases

– Compiler errors (CE) 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.
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 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 compilation or interpretation tool by a more student-friendly alternative. An example is the interpreter used in BIP [10], which generates extensive error messages in understandable language. Figure 2 shows an example of this feedback, which is supplemented by the feedback in Figure 3 if the student asks for more help.

– Solution errors (SE) 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 alterna-

```
*10 LET I = 1

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

Figure 2: Feedback on syntax error in BIP

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

Figure 3: More feedback on syntax error in BIP

tive algorithm that is not accepted. AutoLEP [144] describes the results of matching the student program with several model programs, comparing aspects such as size, structure, and statements.
AnalyseC [148] provides detailed feedback on solution errors by identifying incorrect statements and showing the correct statement from a model solution, shown in Figure 4.

```
Score is : 82
model program Line 8{ s->next = L ->next;}
student program Line 11{ p ->next = s;}
The statement above in student program is incorrect.
model program Line 9{ L ->next = s;}
student program Line 12{ s->next = p ->next;}
The statement above in student program is incorrect.
```

Figure 4: Solution feedback in AnalyseC

ELP [135] (Figure 5) matches a model solution with the student solution at a slightly higher level.

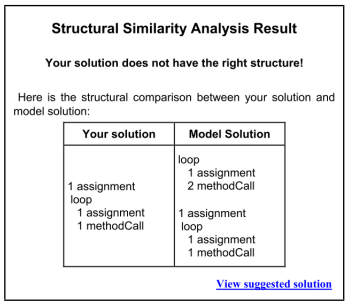

Figure 5: Solution feedback in ELP

– Style issues (SI), such as untidy formatting, inconsistent naming or lack of comments, are not serious mistakes that affect the behaviour of a program. However, many teachers consider learning a good programming style important for novice programmers.
Figure 6 shows the feedback generated by the tool of Jackson00 [72].

– Performance issues (PI). A student program takes too long to run or uses more resources than required. Naur64 [108], one of the earliest systems, checks one particular exercise

```
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)
```

Figure 6: Fragment of style feedback

that lets students 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.

Knowledge about how to proceed (KH)

We identify three labels in KH.

– 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. JITS [131] gives feedback on fixing typing errors, such as:

Would you like to replace smu *with* sum*?*

Proust generates an elaborated error report containing hints on how to correct errors. The following fragment [75] provides such hints:

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 (Figure 5) and AnalyseC (Figure 4) in the KM-SE category also contain the correct code of the solution, therefore we label these tools with KH-EC as well.

– Task-processing steps (TPS). A TPS hint contains information about the next step a student has to take to come closer to a solution. The Prolog tutor Hong04 [68] provides a guided programming phase. If a student asks for help in this phase, the tutor will respond with a hint on how to proceed and generates a template for the student to fill in:

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.

Each of these types of feedback has a level of detail: a hint (◐) that may be a 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 (●).

We expected to find hints on how to *improve* a solution, such as improving the structure, style or performance of a correct solution. We have not found these kind of hints so far. Style- or performance-related feedback is mostly presented in the form of an analysis, which we encode with a KM label.

Knowledge about meta-cognition (KMC)

We have only found one example of KMC so far. HabiPro [141] provides a 'simulated student' that responds to a solution by checking if a student really knows why an answer is correct.

Results

We have found KTC (knowledge about task constraints) feedback in 16% of the tools, KC (knowledge about concepts) feedback in 12%, and KMC (knowledge about meta-cognition) feedback in 1% of all tools. KM (knowledge about mistakes) is by far the largest type, we have found this type in all but one tool. The subtype of KM we have found the most is TF (test failures) in 77% of the tools, after that SE (solution errors, 42%), CE (compiler errors, 36%), SI (style issues, 17%) and PI (performance issues, 9%). We have found KH feedback (knowledge about how to proceed) in 32% of tools, of which 26% give EC feedback (error correction) and 14% give TPS feedback (task-processing steps).

In 59% of the tools only one of the five main types of feedback is given.

4.2 RQ2 Technique

We distinguish general ITS techniques that are not specific for the programming domain, techniques specific for programming. Each category has several subcategories. We do not consider techniques that are used for calculating final grades.

General ITS techniques

– Tools that use model tracing (MT) generate feedback on the process that the student is following. Student steps are compared to production rules and buggy rules [105]. Classic tools use a production system, however, some tools use a slightly different approach. The LISP tutor [36] 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 7.

(1) IF the goal is to define a function called *name*
that accepts *n* arguments and performs the task *process*
THEN code a call to ***defun***
and set subgoals to code
(a) the function name *name*
(b) a list of *n* parameters
(c) the process *process*

Figure 7: A production rule

– Constraint-based modelling (CBM). This technique 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 [105]. INCOM is the only tool we have found so far that uses this technique. The authors of INCOM argue that CBM has its limitations in the domain of programming because of the large solution space for programming problems [88]. 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.

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. Test cases may be predefined or have to be supplied by the student him- or herself. We have also noticed the use of *reflection* in multiple tools, a technique specific for the popular Java language. Reflection can be used to dynamically inspect and execute code. AutoGrader [65] is a lightweight framework that uses Java reflection to execute tests for grading and creating feedback reports.

– Basic static code analysis (BSA) analyses a program (source code or byte code) 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 [128]. Some tools use static analysis for calculating metrics, such as the cyclomatic complexity or the number of comments. The GAME-2 tool performs static analysis by examining comments in a solution [100]. 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 [112].

– Program transformations (PT) transform a program into another program in the same language or a different language. Transformations are often used together with static code analysis to match a student program with a model program. We have found several program transformation techniques in tools for learning programming:

  – Normalisation: transformation into a sublanguage to

decrease syntactical complexity. The technique used in SIPLeS-II [149] is a notable contribution to this field. 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, SPV 6 'different control structures' is handled by transformations that standardize control structures, and SPV 9 'different redundant statements' by dead code removal. As a result, a larger number of student programs can be recognised.

- Migration: transformation into another language at the same level of abstraction. The INTELLITUTOR [136] 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 byte code, is another program transformation technique. We have not found this technique in tools other than compilers, and the external tool FindBugs[2]. FindBugs translates Java code into bytecode, and performs static analysis on this bytecode to identify potential bugs.

- 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 may be borderline cases. Compared to CBM, IBD provides a more complete representation of a solution, that captures the chosen algorithm. The term intention-based diagnosis was introduced by Johnson and Soloway [76] for their tutor PROUST. PROUST has a knowledge base of programming plans, that are implementations of programming goals. One programming problem may have different goal decompositions. Figure 8 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.

SENTINEL READ-PROCESS REPEAT PLAN

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

Figure 8: Simplified plan in PROUST

- 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, we do not use this label.

[2]http://findbugs.sourceforge.net

We have found a number of static analysis tools, for example CheckStyle[3] for checking code conventions, FindBugs[4] for finding bugs, and PMD[5] for detecting bad coding practices. These tools do not specifically focus on novice programmers and may produce 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.

Other techniques

Tools use various A.I. techniques, such as natural language processing or machine learning. The ProPL tutor [85] engages 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 [98] 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'.

We expect that some of the techniques in this category will develop into their own category. For example, we have noticed the use of data analysis in quite a number of recent publications: large data sets with student solutions to exercises are used to generate feedback. Our coding is not final and will evolve while new techniques emerge.

Results

Automated testing is the technique used the most for generating feedback, namely in 78% of the tools. In many tools it is the only technique (in 42%), sometimes it is combined with static analysis. Other tools use testing as a 'last resort' if the tool cannot recognise what the student is doing otherwise. We have found that 9% of the tools use model tracing, and only one (1%) uses constraint-based modelling. Of all tools, 26% use static analysis and 23% use program transformations. Intention-based diagnosis is used in 10% and 10% of the tools use an external tool. We have found various additional techniques in 6% of the tools.

4.3 RQ3 Adaptability

Which input to a tool can be adapted by teachers without recompiling the tool? Using such input a teacher constructs a new exercise or influences the generated feedback, without too much effort or specialised knowledge. Input can take various forms, such as (annotated) code, text, XML, databases, or a custom format. We do not consider input such as marking schemas.

- Solution templates (ST) (e.g. skeleton programs and projects) presented to students for didactic or practical purposes as opposed to technical reasons such as running the program.
  Solution templates are often used for class 2 exercises. An example is the ELP system [135] shown in Figure 9, in

[3]http://checkstyle.sourceforge.net
[4]See footnote 2
[5]https://pmd.github.io/

which students fill in gaps in a Java template with code fragments.

```
        //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();
     }
}
```

Figure 9: Fill-in-the-gap exercise in ELP

Solution templates found in test-based assessment tools are project skeletons, or an interface definition for a data structure that prescribes the names of functions, parameters and return values.

– Model solutions (MS). Correct solutions to a programming exercise are used in many tools. In dynamic analysis they are used for running test cases to generate the expected output. In static analysis the structure of a correct solution is compared to the structure of a student solution.

– Test data (TD), by specifying program output or defining test cases. In older tools testing is done in scripts that are largely the system itself, for example in Naur64. Figure 10 shows a small fragment of its test code [108].

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

Figure 10: Test code in Naur64

In EduComponents JUnit tests can be specified through a web-based interface for creating programming exercises, shown in Figure 11 [5].

– Error data (ED), such as bug libraries, buggy solutions, buggy rules and correction rules. Error data usually specify common mistakes for an exercise, and may include corresponding solutions. In the AA system Chen04 [29] students write test code themselves for some exercises. The instructor provides incorrect solutions to check if the students code can expose them as being erroneous.

– Other. In CourseMarker [66] teachers can configure how much feedback should be given. Some tools let a teacher define custom feedback messages. In Ask-Elle

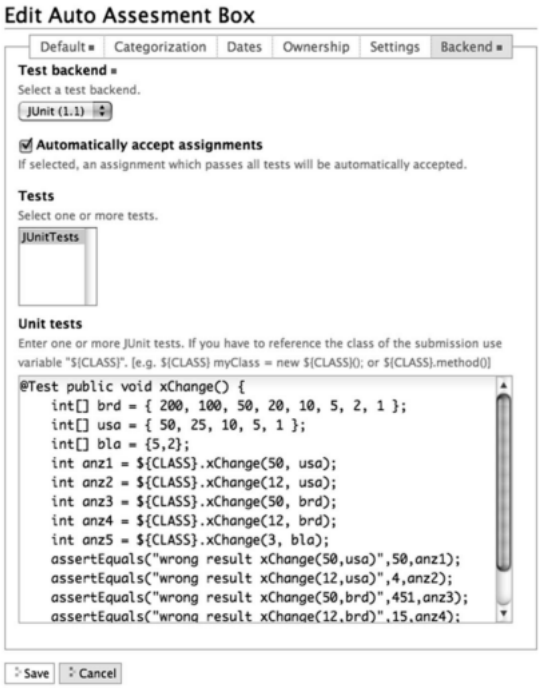

Figure 11: Test editing for EduComponents

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.

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.

Results

We have found that test data is used the most: it is found in 71% of the tools. After that, 49% of the tools use model solutions. Of all tools, 23% offer solution templates, and in 3% error data can be specified. We have found the use of a student model for generating feedback on solutions in 4% of the tools. In 19% of the tools we have found other input.

4.4 RQ4 Quality

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) assessment is based on the experiences and observations of researchers or teachers with 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) assessment compares the characteristics of a tool to a set of criteria related to usability or

a learning theory. For example, THE LISP TUTOR [36] and JITS [131] 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 [27] is based on the completion strategy for learning programming by Van Merriënboer [139]. 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. The user interface of BOSS [78] is evaluated against a set of guidelines and design principles. Some other tools refer to various learning theories, of which some are specific for the programming domain.

- Empirical assessment analyses qualitative data or quantitative data. We distinguish three types of empirical assessment:
  - Looking at the learning outcome (EM-LO), such as mistakes, grades and pass rates, after students have used the tool, and observing tool use. The nature of these experiments varies greatly. For example, PROPL was evaluated in an experiment with 25 students [85]. The students were either in the control group, that used a simple, alternative learning strategy, or in the group that used PROPL. Students both tool 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 size of the test group also varies greatly.
  - Student and teacher surveys (EM-SU) and interviews on experiences with the tool. 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.
  - Technical analysis (EM-TA) to verify whether a tool can correctly recognise (in)correct solutions and generate appropriate hints. Tool output for a set of student submissions is compared to an analysis by a human tutor. SIPLES-II [149] 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.

Results

We have found that 19% of the tools we have examined only provide anecdotal evidence of the success of a tool, and for 7% of the tools we have not found any assessment at all. Of all tools, 12% have been assessed by an analytical method and 72% by some form of empirical assessment, of which student and teacher surveys are the largest group with 42% of the tools.

Not including anecdotal assessment, 23% of the tools have been evaluated by at least two methods.

5. 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 subquestions, and discuss the relation between these answers.

We have found that feedback about mistakes is the largest category found in tools, with information on test failures as the largest subcategory. 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 or not the output meets the requirements of the exercise. We found feedback on solution errors, feedback on the code itself, in a smaller number of tools, and with varying depth and detail.

Very few tools that support exercises that can be solved by multiple (variants of) strategies give feedback with knowledge on how to proceed. According to Boud and Molloy's definition, these tools lack the means to really help a student. In general, the feedback that tools generate is not that diverse, and mainly focused on identifying mistakes. Exceptions are tools that only offer class 2 exercises, that more often provide feedback with 'knowledge on how to proceed'. A disadvantage is that these tools do not support alternative solution strategies, and may restrict a student in his or her problem solving process.

Many of the tools we have investigated are automated assessment tools, which are often used for marking large numbers of student solutions. If marking is the only purpose, one could conclude that more elaborate feedback is not necessary. However, if we want our students to learn from their mistakes, 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.

We have 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 not easy 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 sometimes 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 and four to eight hours for HOGG. We conclude that teachers cannot easily adapt tools to their own needs, except for test-based AA systems.

To answer the last research question, we have investigated how tools are evaluated. Most tools provide at least some form of evaluation, although for 23% 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 the feedback. Moreover, the quality (e.g. the 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 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.

6. CONCLUSIONS AND FUTURE WORK

We have analysed and categorised the feedback generation in 69 tools for learning programming, selected from 17 earlier reviews. Although our search is not yet complete, we report some findings on the relation of feedback content, technique and adaptability. We observe that very few tools that support C3 exercises give feedback with 'knowledge on how to proceed (KH)'. According to Boud and Molloy's definition, these tools lack the means to really help a student. In general, the feedback that tools generate is not that diverse, and mainly focused on identifying mistakes. We have also found that teachers cannot easily adapt tools to their own needs, except for test-based AA systems.

To complete this SLR, we continue our 'backward snowballing' approach by searching the papers we have found so far for relevant references. Furthermore, because our questions are related to computer science and education, we will search a computer science database (ACM), an educational database (ERIC), and a general scientific database (Scopus). Lastly, we will further analyse the results by relating them to programming concepts that students find difficult from Lahtinen et al. [84], common novice programming errors [101, 4], and human tutoring strategies.

Acknowledgements

This research is supported by the Netherlands Organisation for Scientific Research (NWO).

7. REFERENCES

[1] A. Adam and J.-P. Laurent. LAURA, a system to debug student programs. *Artificial Intelligence*, 15(1-2):75–122, 1980.

[2] K. M. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, 15(2):83–102, 2005.

[3] A. Allevato and S. H. Edwards. RoboLIFT: engaging CS2 students with testable, automatically evaluated Android applications. In *SIGCSE Technical Symposium on Computer Science Education*, pages 547–552, 2012.

[4] A. Altadmri and N. C. C. Brown. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *SIGCSE Technical Symposium on Computer Science Education*, pages 522–527, 2015.

[5] M. Amelung, K. Krieger, and D. Rosner. E-Assessment as a Service. *IEEE Transactions on Learning Technologies*, 4(2):162–174, 2011.

[6] J. R. Anderson. *The architecture of cognition.* Lawrence Erlbaum Associates, Inc, 1983.

[7] J. R. Anderson and E. Skwarecki. The automated tutoring of introductory computer programming. *Communications of the ACM*, 29(9):842–849, 1986.

[8] D. Arnow and O. Barshay. WebToTeach: an interactive focused programming exercise system. In *Frontiers In Education Conference*, pages 39–44, 1999.

[9] B. Auffarth, M. Lopez-Sanchez, J. Campos i Miralles, and A. Puig. System for Automated Assistance in Correction of Programming Exercises (SAC). In *Congress University Teaching and Innovation*, pages 104–113, 2008.

[10] A. Barr and M. Beard. An instructional interpreter for basic. *ACM SIGCSE Bulletin*, 8(1):325–334, 1976.

[11] A. Barr, M. Beard, and R. C. Atkinson. A rationale and description of a CAI program to teach the BASIC programming language. *Instructional Science*, 4(1):1–31, 1975.

[12] A. Barr, M. Beard, and R. C. Atkinson. The computer as a tutorial laboratory: the Stanford BIP project. *International Journal of Man-Machine Studies*, 8(5):567–596, 1976.

[13] S. Benford, E. Burke, and E. Foxley. Learning to construct quality software with the Ceilidh system. *Software Quality Journal*, 2(3):177–197, 1993.

[14] S. Benford, E. Burke, E. Foxley, N. Gutteridge, and A. M. Zin. Early experiences of computer-aided assessment and administration when teaching computer programming. *Research in Learning Technology*, 1(2):55–70, 1993.

[15] S. D. Benford, E. K. Burke, E. Foxley, and C. A. Higgins. The Ceilidh system for the automatic grading of students on programming courses. In *Annual on Southeast regional conference*, pages 176–182, 1995.

[16] J. Bennedsen and M. E. Caspersen. Failure rates in introductory programming. *ACM SIGCSE Bulletin*, 39(2):32–36, 2007.

[17] L. Bettini, L. Cecchi, P. Crescenzi, G. Innocenti, and M. Loreti. An environment for self-assessing Java programming skills in undergraduate first programming courses. In *Conference on Advanced Learning Technologies*, pages 161–165, 2004.

[18] M. Blumenstein, S. Green, S. Fogelman, A. Nguyen, and V. Muthukkumarasamy. Performance analysis of GAME: A generic automated marking environment. *Computers & Education*, 50:1203–1216, 2008.

[19] M. Blumenstein, S. Green, A. Nguyen, and V. Muthukkumarasamy. GAME: A generic automated marking environment for programming assessment. In *Conference on Information Technology: Coding and Computing*, pages 212–216, 2004.

[20] C. Bokhove and P. Drijvers. Digital Tools for Algebra Education: Criteria and Evaluation. *Journal of Computers for Mathematical Learning*, 15(1):45–62, 2010.

[21] J. G. Bonar and R. Cunningham. Bridge: Intelligent

Tutoring with Intermediate Representations. Technical report, Carnegie Mellon University, University of Pittsburgh, 1988.

[22] D. Boud and E. Molloy, editors. *Feedback in higher and professional education: understanding it and doing it well*. Routledge, 2012.

[23] P. Brusilovsky and G. Weber. Collaborative Example Selection in an Intelligent Example-Based Programming Environment. In *Conference on Learning Sciences*, pages 357–362, 1996.

[24] P. L. Brusilovsky. Intelligent Tutor, Environment and Manual for Introductory Programming. *Innovations in Education & Training International*, 29(1):26–34, 1992.

[25] J. Caiza and J. Del Alamo. Programming assignments automatic grading: review of tools and implementations. In *International Technology, Education and Development Conference*, pages 5691–5700, 2013.

[26] M. E. Caspersen and J. Bennedsen. Instructional design of a programming course: A learning theoretic approach. In *Workshop on Computing Education Research*, pages 111–122, 2007.

[27] K. E. Chang, B. C. Chiao, S. W. Chen, and R. S. Hsiao. A programming learning system for beginners - A completion strategy approach. *IEEE Transactions on Education*, 43(2):211–220, 2000.

[28] B. Cheang, A. Kurnia, A. Lim, and W.-C. Oon. On automated grading of programming assignments in an academic institution. *Computers & Education*, 41(2):121–131, 2003.

[29] P. M. Chen. An automated feedback system for computer organization projects. *IEEE Transactions on Education*, 47(2):232–240, 2004.

[30] M. Choy, S. Lam, C. K. Poon, F. L. Wang, Y. T. Yu, and L. Yuen. Design and Implementation of an Automated System for Assessment of Computer Programming Assignments. In *Advances in Web-Based Learning*, pages 584–596, 2008.

[31] M. Choy, U. Nazir, C. K. Poon, and Y. T. Yu. Experiences in using an automated system for improving students' learning of computer programming. In *Advances in Web-Based Learning*, pages 267–272. 2005.

[32] J. Coffman and A. C. Weaver. Electronic commerce virtual laboratory. In *SIGCSE Technical Symposium on Computer Science Education*, pages 92–96, 2010.

[33] A. T. Corbett and J. R. Anderson. Student modeling in an intelligent programming tutor. In *Cognitive Models and Intelligent Environments for Learning Programming*, pages 135–144. 1993.

[34] A. T. Corbett and J. R. Anderson. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User Modelling and User-Adapted Interaction*, 4(4):253–278, 1994.

[35] A. T. Corbett and J. R. Anderson. Locus of Feedback Control in Computer-Based Tutoring: Impact on Learning Rate, Achievement and Attitudes. In *SIGCHI conference on Human factors in computing systems*, pages 245–252, 2001.

[36] A. T. Corbett, J. R. Anderson, and E. G. Patterson. Student Modeling and Tutoring Flexibility in the Lisp Intelligent Tutoring System. In *Intelligent tutoring systems: At the crossroads of artificial intelligence and education*, pages 83–106. 1990.

[37] A. T. Corbett, J. R. Anderson, and E. J. Patterson. Problem compilation and tutoring flexibility in the Lisp tutor. In *Conference on Intelligent Tutoring Systems*, pages 423–429, 1988.

[38] C. Daly. RoboProf and an Introductory Computer Course. *ACM SIGCSE Bulletin*, 31(3):155–158, 1999.

[39] C. Daly and J. Horgan. An Automated Learning System for Java Programming. *IEEE Transactions on Education*, 47(1):10–17, 2004.

[40] R. L. Danielson. *Pattie: An automated tutor for top-down programming*. PhD thesis, University of Illinois at Urbana-Champaign, 1975.

[41] S. Davies, J. Polack-Wahl, and K. Anewalt. A snapshot of current practices in teaching the introductory programming sequence. In *SIGCSE Technical Symposium on Computer Science Education*, pages 625–630, 2011.

[42] F. P. Deek, K.-W. Ho, and H. Ramadhan. A critical analysis and evaluation of web-based environments for program development. *The Internet and Higher Education*, 3(4):223–269, 2000.

[43] F. P. Deek and J. A. McHugh. A survey and critical analysis of tools for learning programming. *Computer Science Education*, 8(2):130–178, 1998.

[44] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing*, 5(3), 2005.

[45] C. Douce, D. Livingstone, J. Orwell, S. Grindle, and J. Cobb. A technical perspective on ASAP - Automated system for assessment of programming. In *CAA Conference, Loughborough*, pages 1–14, 2005.

[46] C. Douce, D. Livingstone, J. Orwell, S. Grindle, J. Wood, and J. Curnock. Automatic assessment of programming assignments. In *ALT-C: exploring the frontiers of e-learning - borders, outposts and migration*, 2005.

[47] S. H. Edwards. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing*, 3(3):1–24, 2003.

[48] S. H. Edwards and M. A. Pérez-Quiñones. Experiences using test-driven development with an automated grader. *Journal of Computing Sciences in Colleges*, 22(3):44–50, 2007.

[49] C. Ellsworth, J. Fenwick, and B. Kurtz. The Quiver system. In *SIGCSE Technical Symposium on Computer Science Education*, pages 205–209, 2004.

[50] G. Fischer and J. W. von Gudenberg. Improving the quality of programming education by online assessment. In *Symposium on Principles and practice of programming in Java*, pages 208–211, 2006.

[51] E. Foxley and C. A. Higgins. The CourseMaster CBA System: Improvements over Ceilidh. In *CAA Conference, Loughborough*, 2001.

[52] X. Fu, K. Qian, K. Palmer, B. Peltsverger, B. Campbell, B. Lim, and P. Vogt. Making failure the mother of success. In *Frontiers in Education*

Conference, pages 1–6, 2010.

[53] R. M. Gagné, L. J. Briggs, and W. W. Wager. *Principles of instructional design*. Holt, Rinehart, and Winston, 1992.

[54] T. S. Gegg-Harrison. *Exploiting Program Schemata in a Prolog Tutoring System*. PhD thesis, Duke University Durham, 1993.

[55] A. Gerdes, J. Jeuring, and B. Heeren. An interactive functional programming tutor. In *Innovation and Technology in Computer Science Education*, pages 250–255, 2012.

[56] A. Gerdes, J. T. Jeuring, and B. J. Heeren. Using strategies for assessment of programming exercises. In *SIGCSE Technical Symposium on Computer Science Education*, pages 441–445, 2010.

[57] M. Goedicke, M. Striewe, and M. Balz. Computer aided assessments and programming exercises with JACK. Technical Report 28, 2008.

[58] M. Gómez-Albarrán. The Teaching and Learning of Programming: A Survey of Supporting Software Tools. *The Computer Journal*, 48(2):130–144, 2005.

[59] O. Gotel, C. Scharff, and A. Wildenberg. Teaching software quality assurance by encouraging student contributions to an open source web-based system for the assessment of programming assignments. *ACM SIGCSE Bulletin*, 40(3):214–218, 2008.

[60] O. Gotel, C. Scharff, A. Wildenberg, M. Bousso, C. Bunthoeurn, P. Des, V. Kulkarni, S. P. N. Ayudhya, C. Sarr, and T. Sunetnanta. Global perceptions on the use of WeBWorK as an online tutor for computer science. In *Frontiers in Education Conference*, pages 5–10, 2008.

[61] P. Gross and K. Powers. Evaluating assessments of novice programming environments. In *ICER International Workshop on Computing Education Research*, pages 99–110, 2005.

[62] M. Guzdial. Programming environments for novices. In S. Fincher and M. Petre, editors, *Computer Science Education Research*, pages 127–154. 2004.

[63] J. A. Harris, E. S. Adams, and N. L. Harris. Making program grading easier: but not totally automatic. *Journal of Computing Sciences in Colleges*, 20(1):248–261, 2004.

[64] J. Hattie and H. Timperley. The power of feedback. *Review of Educational Research*, 77(1):81–112, 2007.

[65] M. T. Helmick. Interface-based programming assignments and automatic grading of Java programs. *ACM SIGCSE Bulletin*, 39(3):63–67, 2007.

[66] C. A. Higgins, G. Gray, P. Symeonidis, and A. Tsintsifas. Automated assessment and experiences of teaching programming. *Journal on Educational Resources in Computing*, 5(3), 2005.

[67] C. A. Higgins, P. Symeonidis, and A. Tsintsifas. The marking system for CourseMaster. *ACM SIGCSE Bulletin*, 34(3):46–50, 2002.

[68] J. Hong. Guided programming and automated error analysis in an intelligent Prolog tutor. *International Journal of Human-Computer Studies*, 61(4):505–534, 2004.

[69] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Koli Calling International Conference on Computing Education Research*, pages 86–93, 2010.

[70] P. Isaacson and T. Scott. Automating the execution of student programs. *ACM SIGCSE Bulletin*, 21(2):15–22, 1989.

[71] D. Jackson. A software system for grading student computer programs. *Computers & Education*, 27(3-4):171–180, 1996.

[72] D. Jackson. A semi-automated approach to online assessment. *ACM SIGCSE Bulletin*, 32(3):164–167, 2000.

[73] D. Jackson and M. Usher. Grading student programs using ASSYST. *ACM SIGCSE Bulletin*, 29(1):335–339, 1997.

[74] J. Jeuring, L. T. van Binsbergen, A. Gerdes, and B. Heeren. Model solutions and properties for diagnosing student programs in Ask-Elle. In *Computer Science Education Research Conference*, pages 31–40, 2014.

[75] W. L. Johnson. Understanding and debugging novice programs. *Artificial Intelligence*, 42(1):51–97, 1990.

[76] W. L. Johnson and E. Soloway. Intention-based diagnosis of novice programming errors. In *AAAI conference*, pages 162–168, 1984.

[77] Joint Task Force on Computing Curricula, ACM and IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, 2013.

[78] M. Joy, N. Griffiths, and R. Boyatt. The BOSS online submission and assessment system. *Journal on Educational Resources in Computing*, 5(3), 2005.

[79] C. Kelleher and R. Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2):83–137, 2005.

[80] H. Keuning, J. Jeuring, and B. Heeren. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *Innovation and Technology in Computer Science Education*, 2016.

[81] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, 2007.

[82] C. Köllmann and M. Goedicke. Automation of Java Code Analysis for Programming Exercises. In *Workshop on Graph Based Tools, Electronic Communications of the EASST*, pages 1–12, 2006.

[83] C. Köllmann and M. Goedicke. A Specification Language for Static Analysis of Student Exercises. In *Conference on Automated Software Engineering*, pages 355–358, 2008.

[84] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen. A study of the difficulties of novice programmers. In *Innovation and Technology in Computer Science Education*, pages 14–18, 2005.

[85] H. C. Lane and K. VanLehn. Teaching the tacit knowledge of programming to novices with natural language tutoring. *Computer Science Education*, 15(3):183–201, 2005.

[86] N.-T. Le and W. Menzel. Problem Solving Process

oriented Diagnosis in Logic Programming. In *Conference on Computers in Education*, pages 63–70, 2006.

[87] N.-T. Le, W. Menzel, and N. Pinkwart. Evaluation of a Constraint-Based Homework Assistance System for Logic Programming. In *Conference on Computers in Education*, pages 51–58, 2009.

[88] N.-T. Le and N. Pinkwart. Adding weights to constraints in intelligent tutoring systems: Does it improve the error diagnosis? In *Towards Ubiquitous Learning*, pages 233–247. 2011.

[89] N.-T. Le and N. Pinkwart. INCOM: A Web-based Homework Coaching System For Logic Programming. In *Conference on Cognition and Exploratory Learning in Digital Age*, pages 43–50, 2011.

[90] N.-T. Le and N. Pinkwart. Towards a classification for programming exercises. In *Workshop on AI-supported Education for Computer Science*, pages 51–60, 2014.

[91] N.-T. Le, S. Strickroth, S. Gross, and N. Pinkwart. A review of ai-supported tutoring approaches for learning programming. In *Advanced Computational Methods for Knowledge Engineering*, pages 267–279. 2013.

[92] J. P. Leal and F. Silva. Mooshak: A Web-based multi-site programming contest system. *Software - Practice and Experience*, 33(6):567–581, 2003.

[93] J. P. Leal and F. Silva. Using Mooshak as a Competitive Learning Tool. In *Competitive Learning Institute Symposium*, pages 1–7, 2008.

[94] Y. Liang, Q. Liu, J. Xu, and D. Wang. The recent development of automated programming assessment. In *Conference on Computational Intelligence and Software Engineering*, pages 1–5, 2009.

[95] C.-K. Looi. Automatic debugging of Prolog programs in a Prolog Intelligent Tutoring System. *Instructional Science*, 20(2-3):215–263, 1991.

[96] S. Lowes. 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, 2007.

[97] C. MacNish. Java Facilities for Automating Analysis, Feedback and Assessment of Laboratory Work. *Computer Science Education*, 10(2):147–163, 2000.

[98] C. MacNish. Machine Learning and Visualisation Techniques for Inferring Logical Errors in Student Code Submissions. In *Conference on Advanced Learning Technologies*, pages 317–321, 2002.

[99] R. Matloobi, M. Blumenstein, and S. Green. An Enhanced Generic Automated Marking Environment: GAME-2. *IEEE Multidisciplinary Engineering Education Magazine*, 2:55–60, 2007.

[100] R. Matloobi, M. Blumenstein, and S. Green. Extensions to Generic Automated Marking Environment: Game-2+. In *Interactive Computer Aided Learning Conference*, pages 1069–1076, 2009.

[101] D. Mccall and M. Kölling. Meaningful Categorisation of Novice Programmer Errors. In *Frontiers In Education Conference*, pages 2589–2596, 2014.

[102] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working Group Reports from ITiCSE*, pages 125–180, 2001.

[103] D. C. Merrill, B. J. Reiser, M. Ranney, and J. G. Trafton. Effective tutoring techniques: A comparison of human tutors and intelligent tutoring systems. *Journal of the Learning Sciences*, 2(3):277–305, 1992.

[104] C. M. Mitchell, E. Y. Ha, K. E. Boyer, and J. C. Lester. Learner characteristics and dialogue: Recognising effective and student-adaptive tutorial strategies. *International Journal of Learning Technology*, 8(4):382–403, 2013.

[105] A. Mitrovic, K. R. Koedinger, and B. Martin. A Comparative Analysis of Cognitive Tutoring and Constraint-Based Modeling. In *User Modeling*, pages 313–322. 2003.

[106] D. Morris. Automatic grading of student's programming assignments: an interactive process and suite of programs. In *Frontiers In Education Conference*, pages 1–6, 2003.

[107] S. Narciss. Feedback strategies for interactive learning tasks. *Handbook of research on educational communications and technology*, pages 125–144, 2008.

[108] P. Naur. Automatic Grading of student's ALGOL Programming. *BIT Numerical Mathematics*, 4(3):177–188, 1964.

[109] J. C. Nesbit, L. Liu, Q. Liu, and O. O. Adesope. Work in Progress: Intelligent Tutoring Systems in Computer Science and Software Engineering. In *ASEE Annual Conference & Exposition*, pages 1–12, 2015.

[110] A. Nguyen, C. Piech, J. Huang, and L. Guibas. Codewebs: Scalable homework search for massive open online programming courses. In *Conference on World Wide Web*, pages 491–502, 2014.

[111] P. Nordquist. Providing accurate and timely feedback by automatically grading student programming labs. *Journal of Computing Sciences in Colleges*, 23(2):16–23, 2007.

[112] E. Odekirk-Hash and J. L. Zachary. Automated feedback on programs means students need less help from teachers. *ACM SIGCSE Bulletin*, 33(1):55–59, 2001.

[113] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin*, 39(4):204–223, 2007.

[114] N. Pillay. Developing intelligent programming tutors for novice programmers. *ACM SIGCSE Bulletin*, 35(2):78–82, 2003.

[115] K. A. Rahman and M. J. Nordin. A review on the static analysis approach in the automated programming assessment systems. In *National conference on programming*, 2007.

[116] H. A. Ramadhan, F. Deek, and K. Shihab. Incorporating software visualization in the design of intelligent diagnosis systems for user programming. *Artificial Intelligence Review*, 16(1):61–84, 2001.

[117] K. A. Reek. The TRY system -or- how to avoid

testing student programs. *ACM SIGCSE Bulletin*, 21(1):112–116, 1989.

[118] J. C. Rodríguez-del Pino, E. Rubio-Royo, and Z. Hernández-Figueroa. A Virtual Programming Lab for Moodle with automatic assessment and anti-plagiarism features. In *Conference on e-Learning, e-Business, Entreprise Information Systems, & e-Government*, 2012.

[119] R. Romli, S. Sulaiman, and K. Z. Zamli. Automatic programming assessment and test data generation a review on its approaches. In *International Symposium in Information Technology*, pages 1186–1192, 2010.

[120] R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. *ACM SIGCSE Bulletin*, 33(3):133–136, 2001.

[121] J. A. Sant. "Mailing it in": email-centric automated assessment. *ACM SIGCSE Bulletin*, 41(3):308–312, 2009.

[122] J. Schwieren, G. Vossen, and P. Westerkamp. Using Software Testing Techniques for Efficient Handling of Programming Exercises in an e-Learning Platform. *The Electronic Journal of e-Learning*, 4(1):87–94, 2006.

[123] S. C. Shaffer. Ludwig: An Online Programming Tutoring and Assessment System. *ACM SIGCSE Bulletin*, 37(2):56–60, 2005.

[124] V. J. Shute. Focus on formative feedback. *Review of Educational Research*, 78(1):153–189, 2008.

[125] J. Sorva, V. Karavirta, and L. Malmi. A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Transactions on Computing Education*, 13(4):1–64, 2013.

[126] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez. Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. *ACM SIGCSE Bulletin*, 38(3):13–17, 2006.

[127] M. Striewe, M. Balz, and M. Goedicke. A Flexible and Modular Software Architecture for Computer Aided Assessments and Automated Marking. *Conference on Computer Supported Education*, 2:54–61, 2009.

[128] M. Striewe and M. Goedicke. A review of static analysis approaches for programming exercises. In *Computer Assisted Assessment. Research into E-Assessment*, pages 100–113. 2014.

[129] H. Suleman. Automatic Marking with Sakai. In *Research conference on IT research in developing countries*, pages 229–236, 2008.

[130] E. Sykes. Qualitative Evaluation of the Java Intelligent Tutoring System. *Journal of Systemics, Cybernetics and Informatics*, 3(5):49–60, 2005.

[131] E. Sykes. Design, development and evaluation of the Java Intelligent Tutoring System. *Technology, Instruction, Cognition & Learning*, 8(1):25–65, 2010.

[132] M. Sztipanovits, K. Qian, and X. Fu. The automated web application testing (AWAT) system. In *ACM Southeast Conference*, pages 88–93, 2008.

[133] G. Thorburn and G. Rowe. PASS: An automated system for program assessment. *Computers & Education*, 29(4):195–206, 1997.

[134] N. Truong, P. Bancroft, and P. Roe. Learning to program through the web. *ACM SIGCSE Bulletin*, 37(3):9–13, 2005.

[135] N. Truong, P. Roe, and P. Bancroft. Static analysis of students' Java programs. In *Australasian Conference on Computing Education*, pages 317–325, 2004.

[136] H. Ueno. A generalized knowledge-based approach to comprehend Pascal and C programs. *IEICE Transactions on Information and Systems*, 83(4):591–598, 2000.

[137] M. Ulloa. Teaching and learning computer programming: a survey of student problems, teaching methods, and automated instructional tools. *ACM SIGCSE Bulletin*, 12(2):48–64, 1980.

[138] A. K. Vail and K. E. Boyer. Identifying effective moves in tutoring: On the refinement of dialogue act annotation schemes. In *Intelligent Tutoring Systems*, pages 199–209, 2014.

[139] J. J. Van Merriënboer and M. De Croock. Strategies for computer-based programming instruction: Program completion vs. program generation. *Journal of Educational Computing Research*, 8(3):365–94, 1992.

[140] A. Venables and L. Haywood. Programming Students NEED Instant Feedback! In *Australasian Conference on Computing Education*, pages 267–272, 2003.

[141] A. Vizcaíno. A Simulated Student Can Improve Collaborative Learning. *International Journal of Artificial Intelligence in Education*, 15:3–40, 2005.

[142] A. Vizcaíno, J. Contreras, J. Favela, and M. Prieto. An adaptive, collaborative environment to develop good habits in programming. In *Intelligent Tutoring Systems*, pages 262–271. 2000.

[143] U. von Matt. Kassandra: The Automatic Grading System. *ACM SIGCUE Outlook*, 22(1):26–40, 1994.

[144] T. Wang, X. Su, P. Ma, Y. Wang, and K. Wang. Ability-training-oriented automated assessment in introductory programming course. *Computers & Education*, 56(1):220–226, 2011.

[145] G. Weber. Episodic learner modeling. *Cognitive Science*, 20(2):195–236, 1996.

[146] G. Weber and P. Brusilovsky. ELM-ART: An Adaptive Versatile System for Web-based Instruction. *International Journal of Artificial Intelligence in Education*, 12:351–384, 2001.

[147] G. Weber and M. Specht. User Modeling and Adaptive Navigation Support in WWW-Based Tutoring Systems. In *Conference on User Modeling*, pages 289–300, 1997.

[148] W. Wu, G. Li, Y. Sun, J. Wang, and T. Lai. AnalyseC: A framework for assessing students' programs at structural and semantic level. In *Conference on Control and Automation*, pages 742–747, 2007.

[149] S. Xu and Y. S. Chee. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering*, 29(4):360–384, 2003.

[150] A. Zeller. Making students read and review code. *ACM SIGCSE Bulletin*, 32(2):89–92, 2000.