Tool for Testing Bad Student Programs

IVAN PRIBELA, DONI PRACNER and ZORAN BUDIMAC, University of Novi Sad

This paper presents an effort to address efficient assessment of less than perfect students' solutions in a semi-automated code assessment process. Automated and semi-automated code assessment has its drawbacks when it comes to the poorly written and often non compiling programs created by beginner students. The usual assessment automation approaches often lead to adaptation of courses and assignments to the assessment process, while we try to avoid such trend and thus improve the students' experience. The solution proposed in this paper focuses on automation of the assessment process itself as opposed to the automation of grading which is the usual approach. This paper presents the concept and design of a tool aimed to support such process as well as presents two case studies that validate the use of the proposed tool.

Categories and Subject Descriptors: K.3.2 [**Computers And Education**]: Computer and Information Science Education—*Computer science education*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms: Management, Measurement, Verification

Additional Key Words and Phrases: code assessment, manual assessment, semi-automated assessment, test-based assessment

1. INTRODUCTION

Nowadays, teachers, especially on computer science classes, are faced with grading an ever increasing number of students' assignments. There are three ways to avoid overburdening them: reducing the number of students, increasing the number of teachers, and transferring some of the load from the teachers to the computers. As it is obvious that the first two approaches are often not feasible, getting help from computers and introducing some form of automatic assessment in everyday practice becomes the only possible way of dealing with this issue in an efficient way. This approach, at least theoretically, also increases objectivity of the grading process.

However, fully automated code assessment, with all the benefits it brings, also has its drawbacks. The major one lies in the fact that not everything can be tested by a machine. Typical examples are finer points of coding style like the correct use of procedures and recursion, which are very hard to catch even by very complex metrics. Unfortunately, many approaches in automated assessment are focusing on automation of grading of all aspects of students' solutions.

This often lead to adaptation of courses and assignments to automated assessment, while the opposite should be prefered. Such trend is inappropriate and can prove to be very unfavorable to students, especially on first year programming courses.

The main problem arises from the fact that the beginner students are still learning how to program. They are not yet knowledgeable and disciplined enough to follow strict and rigid program specifications

This work is partially supported by Ministry of Education and Science of the Republic of Serbia, through project no. III 47003: Infrastructure for technology enhanced learning in Serbia;
Author's address: Ivan Pribela, Doni Pracner, Zoran Budimac, Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia; email: pribela@dmi.rs doni.pracner@dmi.rs zjb@dmi.rs

In: Z. Budimac, T. Galinac Grbac (eds.): Proceedings of the 3rd Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications (SQAMIA), Lovran, Croatia, 19.-22.9.2014, published at http://ceur-ws.org.

that are often required by automated testing systems. These students often struggle while creating even simple programming solutions that barely compile and execute.

Such strict rules for the program format and output are perfectly acceptable for experienced students or applicable in coding competitions. However, it is very hard to use them with beginners within their introductory programming classes or generally with less successful students that have difficulties coping with the basics of compilation and program execution.

This results in a rather harsh treatment of nearly satisfactory solution, where the students can lose unproportionally large amount of points to a trifle error. This can be called extreme objectivity, and is easily removed by manual inspection of the student solution.

We propose a better approach which adapts the automated testing to first year courses, and not vice versa. More precisely our proposal is a semi-automated approach that automates the assessment process itself and leaves the finer points like the grading to the instructor.

The rest of the paper is organized as follows. The second section elaborates on the need of a tool that would support the proposed approach. The third section presents one solution that tries to tackle the described problem while the following section offers two typical example situations that illustrate the applicability of the proposed tool in a real learning environment. Finally, a brief conclusion and plans for further work are given in the final section.

2. RELATED WORK

Automatic and semi-automatic assessment of student programing assignments started as early as 1960. Among the first authors were Hollingsworth [Hollingsworth 1960] and Wirth [Forsythe and Wirth 1965], whose systems were designed for assembler and Algol. Since then many other assessment tools were developed [von Matt 1994] that usually followed modern concepts introduced by new operating systems and new programming languages.

Unfortunately, many such ventures focus on a specific programming language or a specific platform, like the systems described in [Dempster 1998; Hawkes 1998; Joy et al. 2005] that focus on Java programming language. This is in line with the trend of Java being the one of the most widely used programming languages on introductory courses. There is also sparing support for other popular languages like Pascal, C/C++ and Python.

There were though some attempts to create a system for testing that will encompass wider spectrum of programming languages, like the one developed in Python presented in [Amelung et al. 2006]. These automated assessment systems are usually language independent if the assessment is based on comparison of the output.

There were also attempts to bring automated assessment of programming to learning management systems (LMS). The assessment system described in [Botički et al. 2008] addresses assessment of SQL Select queries and short programming assignments in a custom LMS environment, providing support for multiple programming languages. However, that solution is limited to the AHyCo LMS developed by its authors and lacks support for large programming projects.

Most contemporary efforts to address the issue of testing non compiling solutions are Web based and offer only visual inspection, like Assyst [Jackson and Usher 1997], which offers a graphical interface that can be used to direct all aspects of the grading process.

BOSS [Joy et al. 2005] is also a Web-based tool, which supports the whole of the assessment process and does not constrain the teacher to present or deliver their assessment material in any given style. Support for software metrics and for unit testing is covered as well.

Another fully functional assessment system called Testovid [Pribela et al. 2011] in comparison to the mentioned systems is built on Apache Ant and thus is not dependent on any programming lan-

guage or specific building and compilation logic. Furthermore, the system is used in a wide variety of situations and environments, as it is very extensible, modular, and can quickly adapt to new trends. It supports both fully automatic and semi-automatic assessment and can be extended to support manual assessment as well.

In their paper [Ahoniemi and Karavirta 2009], the authors Ahoniemi and Karavirta focus on the manual assessment and analyze the use of a rubrics-based grading tool on larger courses with multiple graders. Their results show that the use of such tools can support objective grading with high-quality feedback with reasonable time usage. They also give some pointers for teachers intending to adopt such tools on their courses.

Auvinen goes a step further in his tool called Rubyric [Auvinen 2011] and presents a tool for rubric-based assessment that helps course staff construct textual feedback for students. The tool allows teachers to create grading templates that specify the evaluation criteria and contain feedback phrases for typical mistakes. Because many students make similar mistakes, large portions of feedback can be constructed from pre-written blocks making it possible to give detailed feedback and thus automate the process of grading with little effort.

Another effort at automating the grading process while keeping the human at the center is described in [MacWilliam and Malan 2013]. On a CS50 introductory course at Harvard University, when students complete programming assignments they have traditionally received feedback from staff in the form of comments via email. Staff reported spending significant amounts of time grading because of bottlenecks that included generating PDF documents and manually emailing feedback to students. As it is always preferable that the staff spend less time on logistics and more time providing feedback and helping students, the authors of the paper set out to improve the efficiency of the grading process by creating a web-based utility through which staff can leave feedback for students. This resulted in a 10% fewer hours per week and 13% fewer minutes per student, even while providing as much or more feedback.

Another similar approach with assessment [DeNero and Martinis 2014] aimed to improve the composition quality of student programs argues that the program structure can be understood effectively only by a person. In their paper authors describe their experiences on manually grading over 700 students by a staff of only 10 human graders. To facilitate this effort, they created an online tool that allows teachers to provide feedback efficiently at scale.

However, another topic to keep in mind with manual computer-based assessment for handling large-scale courses is the assessment granularity. The forms that the assessment takes can vary widely from simple acknowledgement to a detailed analysis of output, structure and code. The study [Falkner et al. 2014] analyses the degree to which changes in feedback influence student marks and persistence in submission. In their work, the authors collected data for over a four year period and for over 22 courses. They discovered that pre-deadline results improved as the number of feedback units increase and that post-deadline activity was also improved as more feedback units were available. Unfortunately, greater granularity also means more load for the teachers thus a balance must be found.

3. THE AUTOMATION SCRIPTS

Based on our own problems and experiences in grading large number of student solutions, a set of helper scripts have evolved over a period of time. These scripts have been in use with an array of different assignments in several programming courses on our Department.

The approach assumes that the student solutions are all stored together in one folder with a subfolder corresponding to each student. The naming of the subfolders and the method used to collect all the solutions, be it an automated submission system or manual collection, is of no importance. Al-

though it is recommended that the subfolders contain the student's name or id this is not required by the scripts.

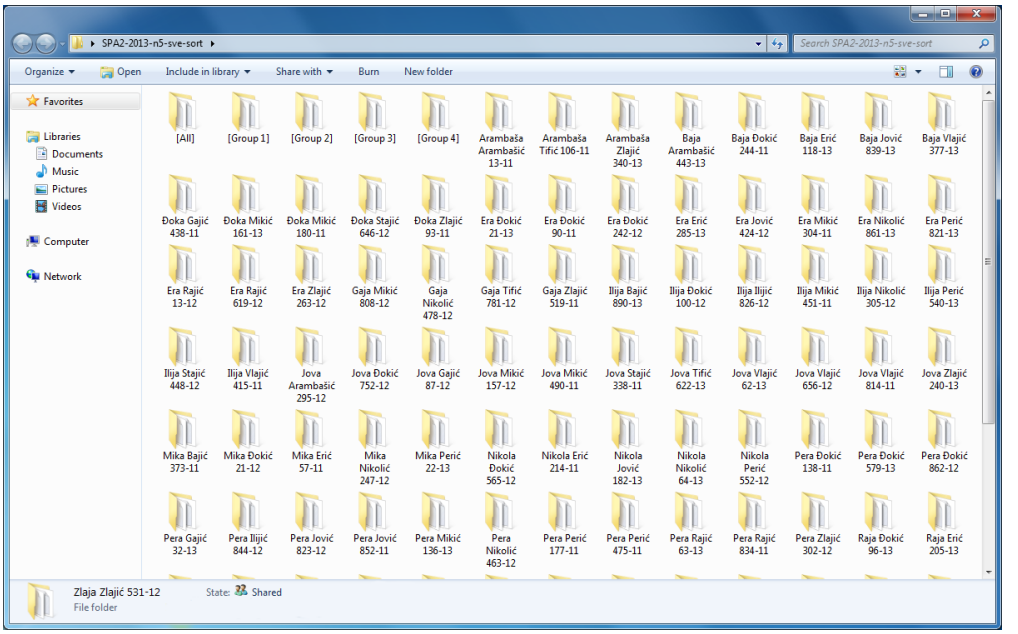

Fig. 1. Folder layout for a typical asignment

A list of the students to be graded is supplied to the scripts via a file usually named "students.txt". This list can be generated from folder names, although most of the courses have some sort of an electronic list of the students, probably in a spreadsheet or exportable as such. Such spreadsheets can also be used for this purpose. Since it possible that the folder names are different than the actual student names, the input can either be a list of the folders or a table with folder identifications and student names and whatever additional data is needed or wanted. In our system we use the student id, their names and groups numbers. A different scenario might also use date and time of last submission, classroom number or whatever might be relevant.

The current implementation of the system uses Apache Ant [Loughran and Hatcher 2007] as the central driver of the process. Ant is a Java library that was originally developed to help with the build process for complex Java projects, with the goal of making something similar to Makefiles for C projects, but at the same time platform independant since it would be running in the Java Virtual Machine. The two basic concepts in an Ant build file are tasks (individual actions such as compilation) and targets (more complex combinations of actions – a number of tasks in order). It provides a number of built-in tasks for compileing, testing and running Java programs, but also to work with filesystems and other types of programs. It can also be easily expanded with custom tasks, and is therefore applicable to any process that is defined in the above terms.

At the core of switching between student solutions to be graded is the custom Ant task that loads up the next solution. The data about the current student being graded is held in a file called "current.properties". The file is in the standard Java properties format, allowing all the data to be accessible to the scripts or other tools as needed. If the file doesn't exist, the assumption is made that the grading has just begun and the first student is to be chosen. Otherwise, the student list is consulted

and the first student after the currently selected one is chosen. Either way the data from the list will be put into the “current.properties”, the content of the temporary folder will be deleted and then populated with files relevant to the next student solution.

The content of this temporary folder consists of combined files from several folders. The script first copies the content from a universal folder with files and data relevant for all students on a course. This folder usually contains basic test files that are not related to the specific group or helper libraries that are used in the assignment. Next, contents of a folder for the corresponding assignment is collected. Most often this folder contains test files for the assignment. Finally the files that represent the student’s solution are copied over to the temporary folder. This final folder stores all files the student has created for the assignment. By default, the script will copy over all the files found in this folder. This behaviour can be overridden and the set of files copied restricted by any regular expression or a set of expressions.

If needed, the list of the included folders can be easily expanded. Apart from the described folders, any files from another source can be included in the content of the temporary folder. These new sources can include supplemental library files, folders with correct solutions, or practically anything that is needed.

Besides helping with the set up of the testing area, the scripts can also help with the actual testing. This help can be in the form of program compilation, using Ant tasks defined for this, running of the student’s solution with different inputs, which may vary depending on the student’s assignment, or basically anything else supported by Apache Ant.

One of the pre-built tasks is used for execution of the student’s program with the simplest input for a quick initial test. This allows the teacher to make manual changes to the program before running the second built-in task that will run all the inputs from the temporary folder that are described by a regular expression. Most of the assignments will also have a task to run the (partial) correct solution for comparison, as well as support to open up the source code of the submitted program and the correct solution or a snippet of code in an editor or difference comparison program. This all allows for saving time on repetitive tasks while giving maximal flexibility in the process for manual interventions.

The advantage with building the system around Apache Ant scripts is that support is widespread, allowing for their usage in many different set-ups, such as Eclipse or NetBeans plugins, a lightweight editor, or even directly from the command line, possibly as part of a larger script. Additionally there are a lot of people familiar with the standard which makes customisations to the scripts much more accessible.

4. TYPICAL USE CASES

To illustrate the proposed approach and the implemented system, we will describe two typical use cases. We have selected assignments from the course Data Structures and Algorithms course on our Department that best illustrate the scripts in action.

The first usage example of the proposed scripts is an assignment that is common in the education of programming – sorting data. The assignment is to load some data from a file, use a specified sorting algorithm and save the data to another file. The students are organised into groups due to spacial limitations, so different groups will have different types of data. An item in the array to be sorted will typically be some form of a record, such as information about people, ie. name/surname/birth year, or an array, such as measurements of temperatures during a week. On this data one of the three possible elementary sorting algorithms should be applied with a simple sorting key, and some additional constraint as a bonus task (for instance sorting by a second criteria).

Automated testing on an assignment such as this proves to be rather difficult for anything other than the fully correct solution. The students often make small mistakes with the indexes in the al-

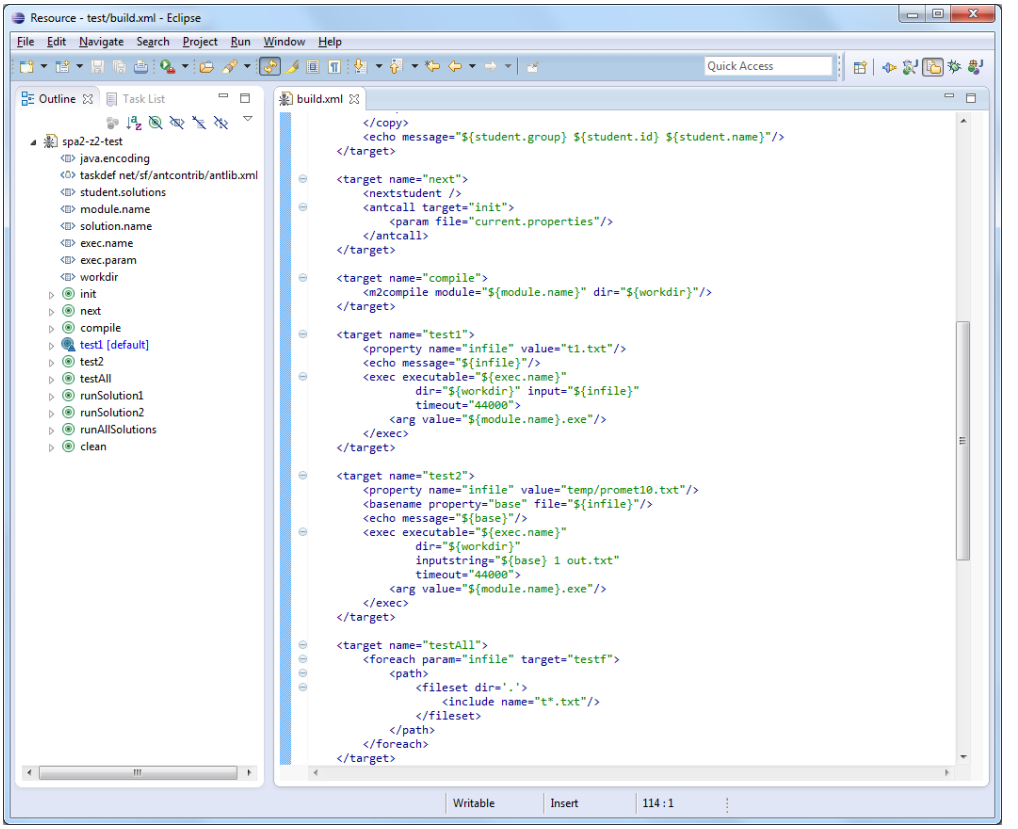

Fig. 2. Script for an asignment loaded into eclipse

gorithms which can result in partially sorted arrays, duplicated items, or even crashes and endless loops. Although these can be very close to the correct solution, an automated tester would give such a solution an unproportionally low amount of points.

Apart from this, it is hard to automatically detect whether a solution that gives the correct result was made with the correctly selected algorithm. The selection of the appropriate sorting algorithm, not just its implementations, is one of the main tasks of this assignment.

Another common problem for automated testing is that students make mistakes in saving the results in the file, forgetting to put in breaks between data or mixing up the order of the fields.

For the second use case we took another assignment common for computer science curricula – searching. Namely the illustrated algorithm is backtracking. One of the assignments that are used on the same course is to find a path through a maze presented in an ASCII text file.

The problem is easy to understand for the students, it illustrates the principles very nicely and it also allows for variance that is required to make different assignments for different groups. For instance there can be “classical” mazes with walls, a starting point and an end point. There can additionally be collectables spread through the maze (pieces of gold or chocolate, radiation levels...) with the task to either maximise or minimise the route in some way (length, collectables collected...). There can be mazes with special rules, such as using only odd numbers to move, or jump in certain directions, different heights that can only be traversed downwards, etc.

This assignment also has the problem of being hard to automatically grade with anything other than fail or pass for a number of inputs. Again there are often mistakes that lead to endless loops or recursion, typos or wrongly defined transitions, all of which are hard to algorithmically detect, and might lead to all of the outputs being wrong, while the code still shows the basic understanding of the problem at hand.

Our tool can help the instructor with these types of submissions in several ways. The testing area is being rebuild for each student from scratch with correct inputs for the student's group. Once a submission has been graded, the next student will be automatically loaded. Both of these save time and increase the reliability of the process.

The instructor can run a number of pre-defined tests on the current submission starting with a very basic test. If it fails there is no use in running the more complex ones straight away, especially if the fault is in the format of the output for instance. Instead the instructor can mark down the detected problem and make manual changes to fix them and then re-run the tests to check for further faults. Correct solutions to the problem can also be run on the same inputs for comparison.

To further help with the grading there are also options to run comparisons to pre-defined blocks of code that contain the correct solutions, either as complete programs, or just snipets that apply to a particular part of the assignment. This can help in detecting subtle mistakes such as array index problems, or special cases that are often overlooked and can pass undetected by an worn instructor. It can also help if there was a prescribed method of solving the problem – as in our sorting use case, where a particular algorithm is part of the assignment.

5. CONCLUSIONS AND FUTURE WORK

The proposed scripts can help greatly with the repetitive tasks during the manual grading. Typical assignments that show the benefits of using such scripts are illustrated in the previous chapters. The teacher only needs to monitor a single folder. The student's submissions will be copied one by one automatically without even a need to manually look up who the next student is let alone where the solution and other files are. The appropriate test cases will be loaded as well, and the grader has an array of options that can be applied as the situation demands. These options can range from an application of simple input files to an executing program, to the comparison of student code with snippets of correct solutions. Also, only one test case can be applied, all of the prepared test cases, or an ad hoc example. All the time during such grading the teachers have the options for manual changes of the students' code as needed and can combine all the mentioned options as they like.

The proposed scripts greatly decrease the time needed to set up and tear down a testing environment for fair grading of solutions. At the same time they also eliminate a lot of the possible sources of mistakes that are common for repetitive tasks.

Our current research efforts are directed at gathering more data on the impact of this approach on the time teachers spend on assessment compared to standard two-phase approach. The experiences in using these scripts for the last few years have been positive.

Another objective is to collect more usage examples and then improve the scripts to cover more scenarios during the manual testing or incorporate the scripts in a larger environment. This is aimed to maximize the help offered to the teachers and further shorten the time needed for the repetitive tasks.

ACKNOWLEDGMENTS

The work is partially supported by Ministry of Education and Science of the Republic of Serbia, through project no. III 47003: Infrastructure for technology enhanced learning in Serbia.

REFERENCES

Tuukka Ahoniemi and Ville Karavirta. 2009. Analyzing the Use of a Rubric-based Grading Tool. *SIGCSE Bull.* 41, 3 (July 2009), 333–337. DOI:http://dx.doi.org/10.1145/1595496.1562977

Mario Amelung, Michael Piotrowski, and Dietmar Rösner. 2006. EduComponents: Experiences in e-Assessment in Computer Science Education. *SIGCSE Bull.* 38, 3 (June 2006), 88–92. DOI:http://dx.doi.org/10.1145/1140123.1140150

Tapio Auvinen. 2011. Rubyric. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research (Koli Calling '11)*. ACM, New York, NY, USA, 102–106. DOI:http://dx.doi.org/10.1145/2094131.2094152

Ivica Botički, Ivan Budišćak, and Nataša Hoić-Božić. 2008. Module for online assessment in AHyCo learning management system. *Novi Sad J. Math* 38, 2 (2008), 115–131.

Jay Dempster. 1998. Web-based assessment software: Fit for purpose or squeeze to fit. In *Proc. of 2nd Computer Assisted Assessment Conference*.

John DeNero and Stephen Martinis. 2014. Teaching Composition Quality at Scale: Human Judgment in the Age of Autograders. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 421–426. DOI:http://dx.doi.org/10.1145/2538862.2538976

Nickolas Falkner, Rebecca Vivian, David Piper, and Katrina Falkner. 2014. Increasing the Effectiveness of Automated Assessment by Increasing Marking Granularity and Feedback Units. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 9–14. DOI:http://dx.doi.org/10.1145/2538862.2538896

George E. Forsythe and Niklaus Wirth. 1965. *Automatic Grading Programs*. Technical Report. Stanford University, Stanford, CA, USA.

Trevor Hawkes. 1998. An experiment in computer-assisted assessment. In *Proc. of 2nd Computer Assisted Assessment Conference*.

Jack Hollingsworth. 1960. Automatic Graders for Programming Classes. *Commun. ACM* 3, 10 (Oct. 1960), 528–529. DOI:http://dx.doi.org/10.1145/367415.367422

David Jackson and Michelle Usher. 1997. Grading Student Programs Using ASSYST. *SIGCSE Bull.* 29, 1 (March 1997), 335–339. DOI:http://dx.doi.org/10.1145/268085.268210

Mike Joy, Nathan Griffiths, and Russell Boyatt. 2005. The Boss Online Submission and Assessment System. *J. Educ. Resour. Comput.* 5, 3, Article 2 (Sept. 2005). DOI:http://dx.doi.org/10.1145/1163405.1163407

Steve Loughran and Erik Hatcher. 2007. *Ant in Action* (second ed.). Manning Publications. 600 pages. http://antbook.org/

Tommy MacWilliam and David J. Malan. 2013. Streamlining Grading Toward Better Feedback. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13)*. ACM, New York, NY, USA, 147–152. DOI:http://dx.doi.org/10.1145/2462476.2462506

Ivan Pribela, Mirjana Ivanović, and Zoran Budimac. 2011. System for Testing Different Kinds of Students' Programming Assignments. In *Proceedings. of 5th International Conference on Information Technology ICIT*.

Urs von Matt. 1994. Kassandra: The Automatic Grading System. *SIGCUE Outlook* 22, 1 (Jan. 1994), 26–40. DOI:http://dx.doi.org/10.1145/182107.182101