Computer Aided Grading with Agar

Titus Winters and Tom Payne, UC Riverside

***Abstract*— Computer-based grading tools have existed for nearly as long as computing courses. The majority of these tools have focused on completely automatic grading of functional requirements, leaving no room for subjectivity, and generally eschewing human feedback in favor of total automation. We argue that these tools are of little practical use because they severely limit the types of assessments that can be graded, and force the user to adopt to the paradigms of the grading tool, rather than vice versa. We present Agar, a tool designed to compensate for those possible shortcomings, discuss the design of Agar, and discuss unanticipated usage patterns that have already sprung up in our user base.**

I. Introduction

Given computer scientists's fondness of automate repetitive tasks, it comes as no surprise that computer-aided assessment (CAA) has been used continuously in various forms for nearly 50 years. Given a set of student submissions, and the repetitive tasks of checking compilation, program correctness, coding style, and documentation, it is hard to imagine a computer science instructor that hasn't thought of some automation scheme during long and tedious grading sessions.

However, most CAA systems seem to be lacking in two areas, which prevents them from becoming very popular. First, especially in Computer Science, CAA systems generally focus on automated testing of code and little else. A relatively common example of this is to use JUnit to perform method-level regression testing of Java assignments [3]. Programs can also be specified in the now-familiar pattern of ACM Collegiate Programming Contest problems: given a precise input and output specification, test that the student submissions function as a precise transformation on the input, and grade using *diff(3)*. Systems like [6], [1], and countless others have relied on this level of precision, and performed strictly functional grading.

A second major hurdle barring widespread adoption of CAA systems is usability. While the academic community undoubtedly has thousands of CAA systems in the literature or active deployment at various institutions, nothing has grabbed the community as a real "killer app." One reason for this may be paradigm mismatch: although there has been a lot of time invested in CAA systems, we collectively have much more experience with the "red-pen" style of grading. In order to be really usable and have a chance of catching on, a CAA system needs to match existing paradigms of usage to the greatest extent possible, allow as many real-world usages as possible, and restrict the user as little as possible.

This paper introduces Agar, a CAA system that appears to be unique in the literature by overcoming these issues. Agar has a fully featured functionality-testing system for checking compilation, input and output, code unit testing and more. However, every test that Agar performs can be overruled by the human grader, and Agar's usefulness extends well beyond objective code testing. Agar has been used nearly a hundred subjective assignments, quizzes, and exams. It has also been granted credit for speeding up grading of tests for large intro-level classes by at least a factor of two.

This paper is organized as follows: Section II provides a brief history of the 47 documented years of CAA in the literature, highlighting common design themes and what we regard as overlooked design features. A description of the design and features of Agar is given in Section III. Finally, Section V summarizes the contributions of this tool.

II. History of CAA

Computer-aided assessment has a history dating at least as far back as 1959. That year a computer program was used to test machine-language programming assignments, and was publicized the following year[2]. This paper by Jack Hollingsworth of Rochester Polytechnic Institute captures the very essence of CAA results since then: Given 120 students in a full-semester programming course, he claims "We could not accommodate such numbers without the use of the grader." This is a feeling that has surely been echoed thousands of times as enrollments in CS have surged upward again and again over the past four decades. Surely a sizable number of CAA systems have seen their start because of a similar thought.

Security is presented as a significant concern for Hollingsworth: the computer systems being utilized did not provide a method for protecting the grader in memory, so malicious or buggy student submissions could alter the grader itself.

The Hollingsworth grader was based on the paradigm of precise functional grading wherein programs are completely specified, and only a complete functional match is considered a correct submission, which stifles many opportunities for student creativity. Hopefully recent work like [4] will convince the community that this may be a good paradigm to practice from time to time to ensure students are capable of following directions, but to attract a more diverse population and keep student retention high, we must allow for more creativity.

However, this theme of I/O-spec grading continues through most or all of the CAA systems that are in the literature. In the late 1960s, the breakthrough CAA system was the Basser Automatic Grading Scheme (BAGS) [1] from the University of Sydney, which represented the first time that a grader could be used without special support from a human operator. The paper describing BAGS makes it quite clear what types of programs are acceptable for grading: precise mathematical operations with zero margin for error. Useful, challenging assignments undoubtedly, but not the type of assignment that captures the interest of students currently being drawn away to other majors.

The mid-1990s saw the development and publication of systems like Kassandra[6], an automated grading system designed for a scientific computing course to evaluate Maple or Matlab code. Kassandra is a system with a number of very impressive features, like networked operation and a secure client-server paradigm that absolutely prevents students from interacting with the reference solutions. Especially in the face of growing sophistication on the part of the students, Kassandra certainly represents a step in the right direction: who hasn't worried at some point about compiling and running student code, either with the use of a CAA tool or manually? Still, we find that Kassandra focuses exclusively on grading precise functional behavior, and little more.

One aspect of the adoptability of these systems that cannot be easily evaluated from the published literature is the usage paradigm for the system. A fundamental concept in human-computer interaction is that programs are difficult to use if the system model doesn't match with the user model [5]. Since users generally come to a new program with little or no initial concept of the precise workings of the system, user model is often wildly erratic, at least during the initial stages of use. Users experience a great deal of stress when they are suffering from such a model mismatch. A reliable method for reducing the time needed to correct the user model is to leverage existing, known usage patterns. In essence, the system model should match existing usage patterns as much as possible so that a user doesn't have to deal with expectation violations that cause a program to be "hard to use." Without having actually interacted with these other CAA systems, we cannot judge for certain whether the program matches existing grading styles, but there doesn't appear to be much reason to assume that this sort of usability was of much import to the designers of these systems.

III. AGAR

Agar attempts to be a highly usable tool by focusing on a handful of design principles:

1) Allow existing grading styles - Both in terms of how an individual grader grades, and how grading tasks are broken up for large classes, Agar attempts to match existing grading styles.
2) Support the subjective - Agar goes beyond most CAA systems by supporting grading of subjective features (style, documentation, written work), rather than just functionality. This means it can be used for tests and quizzes, or programming work with a creative aspect.
3) Eliminate repetition - Whenever possible, Agar eliminates the repetitive tasks in grading. One of the primary ways it does so is by making student feedback instantly reusable. But this trait also applies to recording scores in a gradebook, emailing students their scores and feedback, and more.

Grading with Agar can be broken up into five main phases: identifying submission files, setting up a rubric, running tests and conversions, performing any needed manual grading, and reporting scores.

A. Identifying Submissions

Our homework-submission system has students turn in an entire directory at once. One effect of this is that there are often extra files that are not strictly necessary to make a programming assignment function: object files, backup source files, sample inputs, etc. Agar includes three main methods of identifying which of the files in a student's submission directory are actually part of their submission and should be included in Agar's list of files for that student. When we first started working on CAA systems, one of our requirements was that students submit files with a precise filename. On average, ten to twenty percent of the students ignored these requirements, regardless of how terrible a penalty we threatened. Eventually we decided it would be far better for our systems to be able to identify which files were pertinent automatically.

The default method is using filemasks to identify which files are important. When starting a new Agar workspace it prompts the user for what kind of grading is to be done. For written grading, this sets the filemask to **.ps, *.pdf, *.txt, *.doc*. Any filename in the submission directory matching *any* of those Unix globs will automatically be included in the student's file list. Similarly for C/C++ grading, the filemask is set to **.c, *.cc, *.cpp, *.h, *.hh, Makefile*. This covers the majority of our grading, but an option exists to manually enter an arbitrary filemask as well. Once the filemask is identified, a base directory is selected. It is assumed that this directory contains student submissions in the format generated by our turnin system (this can be easily altered for other environments). For each submission directory, the filemask is applied and a collection of matching files are extracted. If no files match the filemask in one or more directories, then the user is automatically prompted to identify submission files manually for those submissions.

Another method of identifying files relies on the student submitting a valid (but simple) Makefile. For each dependency in the Makefile that does *not* have an associated creation rule, if that file exists in the directory it is added to the file list. The Makefile parser also will do simple variable expansions, although it does not fully emulate Make and its handling of implicit dependencies and creation rules: all files must be explicitly mentioned to be included.[1]

In general, one of the above two methods will suffice to identify the vast majority of submission files. Once in a while a student will name a file oddly or will have some other problem with a submission, and it is necessary to manually identify the files for that submission. Any possible submissions in the base directory that cannot be understood with one of the above methods can be identified by hand simply by opening a file browser in that directory and manually adding files to the submission (Figure 1).

Usually, identification of submission files is completely automatic, and requires less than five minutes for a moderately sized class, even in the presence of misunderstood submissions.

[1]This means that, for example, implicit rules for generating *.o* files will not be picked up by the system.

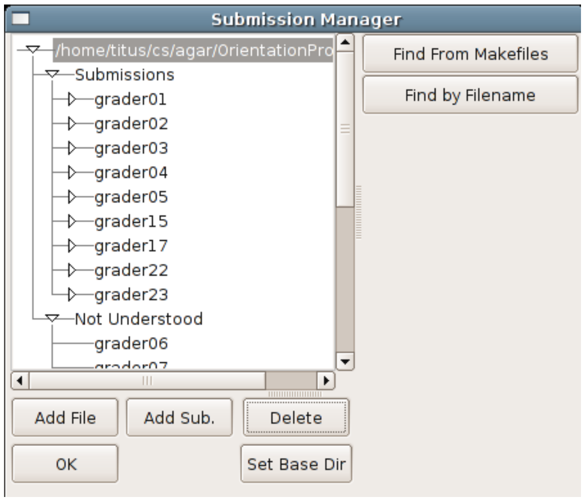


Fig. 1. Submission Manager

B. *Creating a Rubric*

Once submission files are identified, the next step is to create the basic grading rubric. In Agar, a rubric is a tree of tests. The top-level items are the point categories that will be reported for student grades. For programming work these are often one major category ("Insert", "Remove", "Style", etc), for written work there is generally a top-level rubric item for each question. Each rubric item is selected from the available set of tests (the left portion of Figure 2). Additionally, if a rubric item fails, any sub-tests of that item will be executed, allowing for conditional execution when necessary. The most common usage of this hierarchical testing thus far is for compilation: in our department we generally require the *g++* flags *-W -Wall -Werror -pedantic* for lower-division programming courses. This helps ensure that the students are writing good code. Often there will be a mandatory penalty for code that fails to compile with these flags: by allocating some rubric points to the default "Compile" (with flags), and then creating a sub-test that re-attempts the compilation without the flags, we can quickly compile all submissions that are compilable as well as identify which students had warnings in their code.

Most of the tests shown in Figure 2 are separate executable programs adhering to the Agar tool specification. With very few basic requirements on parameter parsing and exit codes, Agar users can write their own programs and scripts to perform testing tasks in Agar. The tool specification is quite minimal, requiring on the order of tens of lines of code, and eliminates the need for a complex plugin structure, dynamic linking, shared-object generation, or any other complex scheme. Anybody should be able to easily write extensions for Agar, in any language, so long as it can read command line parameters, run as an executable under Unix, and return different error codes when terminating.

In addition to determining which tests make up a rubric, it is also important to identify the point values for each item. Each rubric item has an associated point value, which is easily changed using the spin control on the lower-right portion of the screen. It is also possible to create *comments* that are automatically assigned to submissions that pass or fail on a particular test. Tests themselves are fairly limited, and are generally intended to be the first-pass: anything that passes is good, anything that doesn't pass or cannot be automatically evaluated will be checked by hand later.

For an average-sized course, creation of a rubric for written work is nearly automatic, and a suitable rubric for programming work takes five-to-ten minutes.

C. *Running Tests*

Once the rubric is set up, tests can be run on all submissions. One nice feature of Agar is that many of the tests that actually run the student code have builtin safeties to ensure that student code doesn't run too long, use up too much memory, etc. We do not currently resort to sandboxing student programs in a *chroot* jail for security purposes, since we have never encountered student code malicious enough to warrant such measures, but building such a secure execution environment test is certainly possible within Agar's framework.

After running tests, Agar color-codes rubric items that have a perfect pass or fail rate. Green tests indicate every submission passed, red tests indicate everyone failed. Red tests often indicate that something was improperly configured. After re-configuring the test, individual rubric items can be re-run without re-running all tests. Similarly, if a particular student's submission needs to be altered and re-tested, that submission can have all tests re-run without affecting other test results. It may take as long as ten minutes to run all the tests for an average programming assignment, depending on the computational complexity of the assignment and number of students in the class.

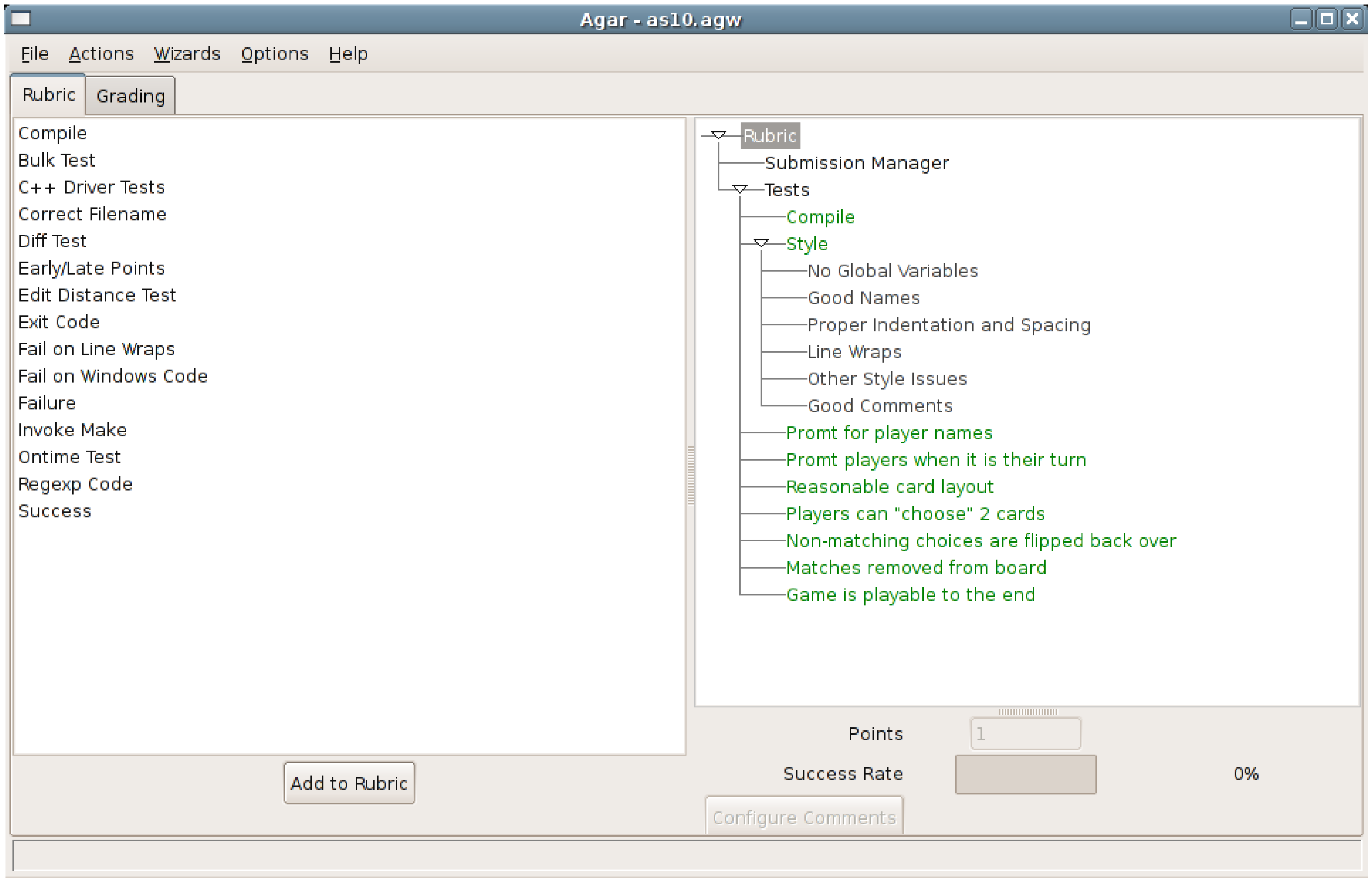
D. *Manual Grading*

When testing is complete, the grader can switch to the "Grading" view (Figure 3). From there they can switch between submissions, view individual submission files for each submission, launch a terminal in a particular submission directory, and most importantly, assign comments.

Comments are used in Agar just like comments are used when grading with a red-pen: identify something worth commenting on, provide the student some feedback, and possibly associate a point value with it. Agar comments can be positive or negative, additive or multiplicative (the difference between bonus points and bonus percentage), or can simply set the value for a particular top-level rubric item. Comments can also be set to apply to all rubric items, like in the example shown in Figure 4, which shows a comment that gives the student a zero on the entire assignment.

The general process for grading in this phase is to go through each submission and check anything needing to be manually investigated. For well-specified programming assignments with no Style component to their scores, this could be nothing at all, since Agar does support total automation. For written assignments or tests, this is the primary grading task.

In practice, one of the most useful pieces of functionality in Agar is the "write-once" comment system. Since a large class is almost guaranteed to repeat errors over and over, this

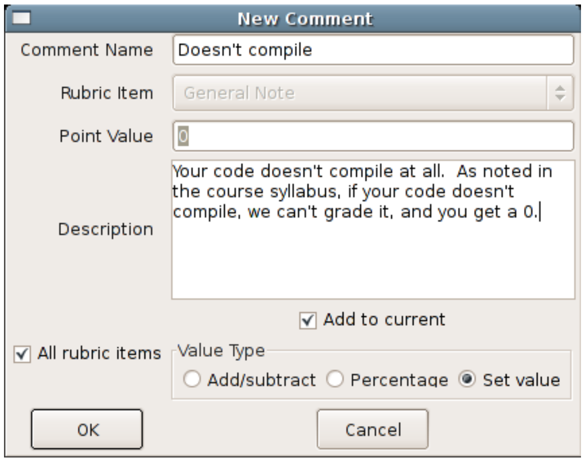


Compile
Bulk Test
C++ Driver Tests
Correct Filename
Diff Test
Early/Late Points
Edit Distance Test
Exit Code
Fail on Line Wraps
Fail on Windows Code
Failure
Invoke Make
Ontime Test
Regexp Code
Success

```
Rubric
├─Submission Manager
└─Tests
  ├─Compile
  ├─Style
  │ ├─No Global Variables
  │ ├─Good Names
  │ ├─Proper Indentation and Spacing
  │ ├─Line Wraps
  │ ├─Other Style Issues
  │ └─Good Comments
  ├─Promt for player names
  ├─Promt players when it is their turn
  ├─Reasonable card layout
  ├─Players can "choose" 2 cards
  ├─Non-matching choices are flipped back over
  ├─Matches removed from board
  └─Game is playable to the end
```

Points: 1
Success Rate: 0%

Fig. 2. Rubric Creation



Comment Name: Doesn't compile
Rubric Item: General Note
Point Value: 0
Description: Your code doesn't compile at all. As noted in the course syllabus, if your code doesn't compile, we can't grade it, and you get a 0.
[✓] Add to current
[✓] All rubric items
Value Type: Add/subtract Percentage [✓ Set value]

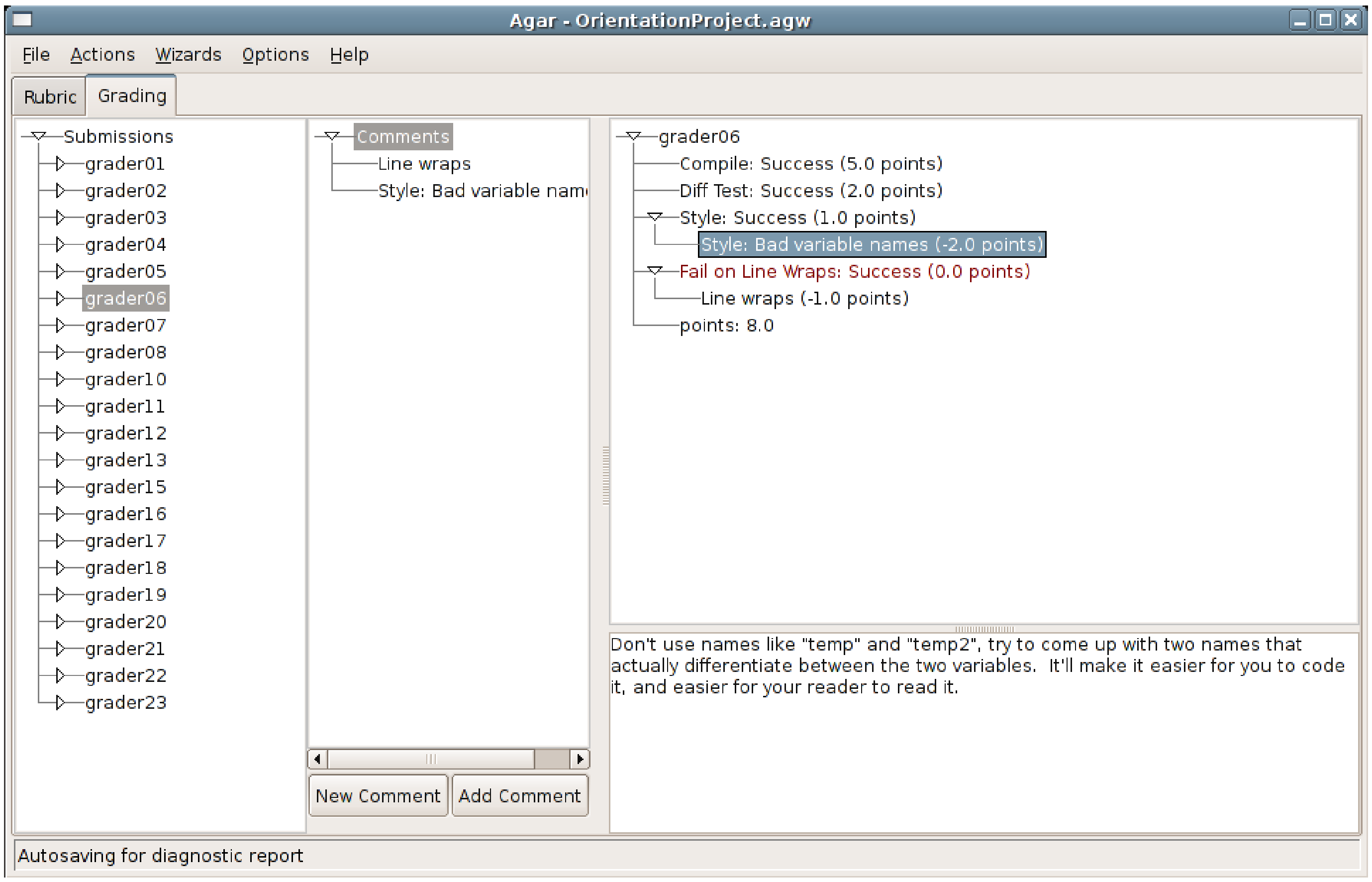
Fig. 4. Creating or Editing a Comment

commenting system can be a great time saver for a number of reasons. First of all, many people type faster than they write, so typing out feedback is a good step. Secondly, since comments are saved for re-use, when a duplicate error is encountered, the comment can be dragged from the list and dropped on the duplicate error in the new submission, saving even more time. Third, comments are assigned by reference, and thus if they are edited later all instances of that comment are updated. This is useful in cases where penalties need to be adjusted later on, or feedback requires editing before it is sent back. All of these features greatly reduce the amount of time needed to do a fair job of grading, and increase the overall consistency of the grading itself. Since it is easier to assign an existing comment regarding a problem than to create a new comment, there is a tendency to be more fair in giving the same penalty for similar problems. Students appreciate this consistency.

E. *Annotation*

There is also an alternate interface for assigning comments. Especially for written work, tests, or quizzes, it is often best if comments can have some “locality”, like “Notice that this node in your 2-3 Tree has 1 child, which is forbidden in a 2-3 Tree.” A comment of this type makes more sense when placed on the page near the offending node of the tree. Agar’s annotation interface allows many file types, including PostScript, PDF, DOC, plain text, source code, and many image types, to be displayed on screen and annotated with comments (Figure 5).

Using the annotation interface allows for graders to grade either submission-by-submission or question-by-question (especially useful for scanned tests where all questions are on the same page of the submission). When mailbacks are generated for students, all of the “pages” of their submission are converted to PDF, and the localized comments are added to the



```
grader06
  Compile: Success (5.0 points)
  Diff Test: Success (2.0 points)
  Style: Success (1.0 points)
    Style: Bad variable names (-2.0 points)
  Fail on Line Wraps: Success (0.0 points)
    Line wraps (-1.0 points)
  points: 8.0
```

Don't use names like "temp" and "temp2", try to come up with two names that actually differentiate between the two variables. It'll make it easier for you to code it, and easier for your reader to read it.

Autosaving for diagnostic report

Fig. 3. Basic Grading Interface

PDF as annotations viewable within Adobe Acrobat Reader[2] as colored sticky-notes that expand to hold the comment text.

F. Reporting Scores

When the manual grading is completed, the remaining tasks are to generate feedback for the students and to record scores in the course gradebook. Generating feedback for the students is easy, assuming the students all have email addresses in the same domain. Agar identifies student submissions by their username, and can be configured with an outgoing SMTP server and the domain of the students. It then generates mailback files containing score summaries, test results, and comment feedback. If the annotation system was used it also converts the submission to PDF and annotates it with the appropriate comments. When the mailbacks have all been generated, it automatically emails them out to the students.

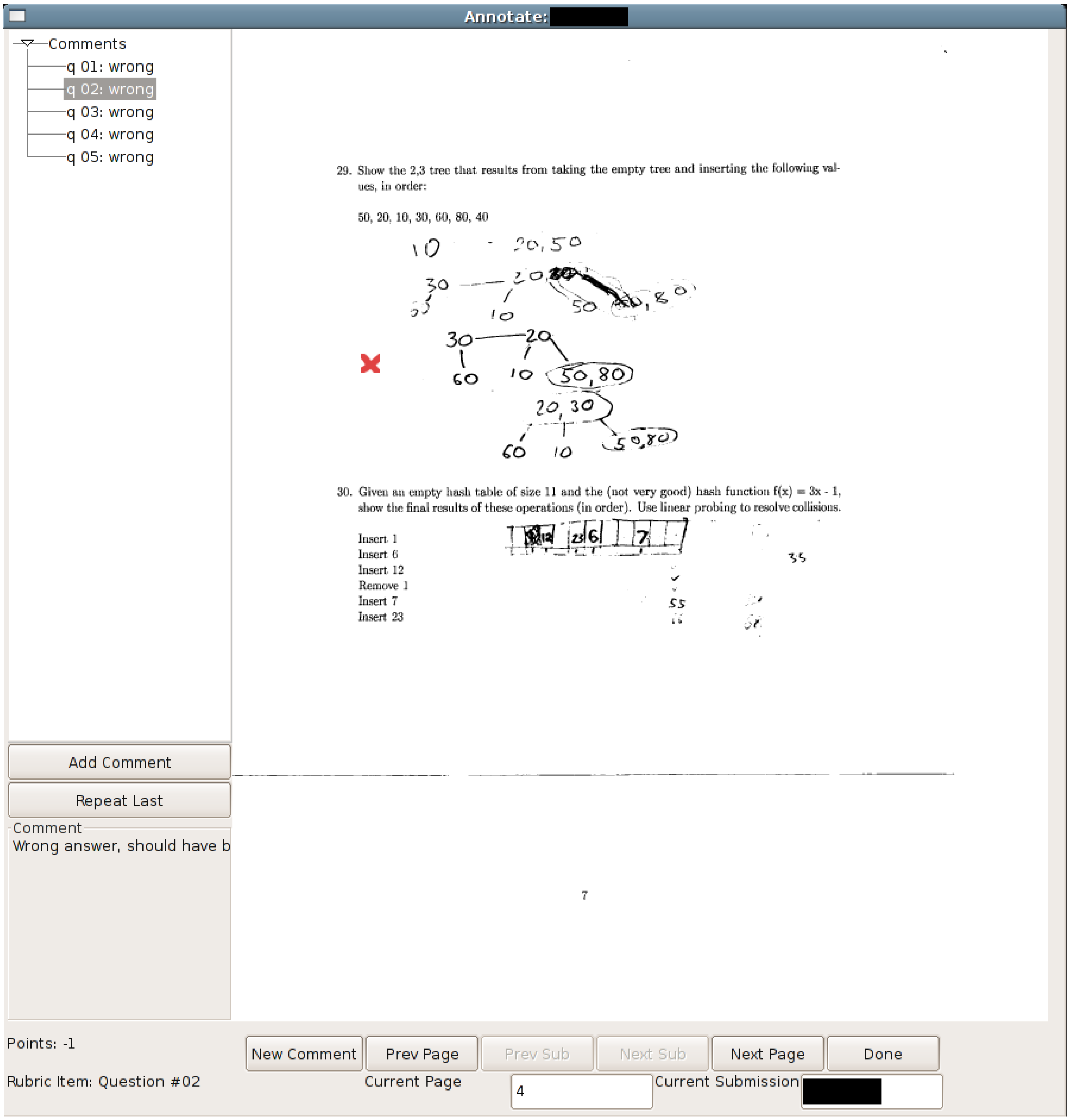
For exporting grades, Agar primarily supports the use of a spread sheet for storing grades. It prompts for the location of the course gradesheet, which can be in either Microsoft Excel format or the open-source Gnumeric format. It then searches the gradebook for a sheet called “Summary” which has a listing of the student usernames. Based on the row-ordering of those usernames, it creates a new sheet with the scores for the current assignment, with one column for each top-level rubric item and with students ordered into the same row order in the new sheet as they appear on the Summary sheet. It also (optionally) will export a column of indirect-references to the total points column of the Summary sheet to keep it up to date. Not only does use of Agar obviate hand entry of score data, it also allows scores to be recorded and tracked at a much finer granularity than would otherwise be kept. This extra data allows better statistics to be calculated, giving interested instructors better insight into what their students actually knew on a given assignment, test, or quiz[7].

[2]Most other PDF viewers ignore annotation information.

IV. Usage Patterns

Two relatively simple features of Agar that have been found to be quite powerful are the ability to save different portions of the workspace into different files, and to merge Agar files together. These abilities allow for usage patterns similar to those that are normally found in courses with large grading loads.

One feature that has been used repeatedly is that of the pre-developed rubric template. The Head TA for a large class generally creates a rubric and a default set of expected comments. The Head TA will then save just the rubric and comments into a “template” file that is given to the other TAs along with a list of which students to grade. Each TA runs Agar on their own set of submissions, grades using that rubric and those comments for even greater consistency, and then sends



Annotate: [redacted]

Comments
- q 01: wrong
- q 02: wrong
- q 03: wrong
- q 04: wrong
- q 05: wrong

29. Show the 2,3 tree that results from taking the empty tree and inserting the following values, in order:

50, 20, 10, 30, 60, 80, 40

30. Given an empty hash table of size 11 and the (not very good) hash function f(x) = 3x - 1, show the final results of these operations (in order). Use linear probing to resolve collisions.

Insert 1
Insert 6
Insert 12
Remove 1
Insert 7
Insert 23

7

Add Comment

Repeat Last

Comment
Wrong answer, should have b

Points: -1

Rubric Item: Question #02

New Comment | Prev Page | Prev Sub | Next Sub | Next Page | Done

Current Page: 4

Current Submission: [redacted]

Fig. 5. Agar Annotation Interface

back an Agar workspace complete with grading information. The Head TA merges all of these Agar workspaces together before sending mailbacks and exporting grades. This is a simple method of splitting the workload student-by-student. After grading, the templates can be updated with the most common problems not covered with the original comments, and saved for use in future course offerings.

When grading exams for large courses, Agar also allows the work to be split question-by-question. Again, a Head TA creates a rubric and possibly some basic comments for each question, and then gives a copy of the full workspace (with all rubric information and student submissions) to each TA, along with instructions for which question to grade. The TAs grade in parallel, just as they would with paper grading, and then send their Agar workspaces back for merging. While the selective saving and merge abilities of Agar are not technically complex, they have proven to be invaluable in the adoption of the system.

V. Conclusions

Agar is, as far as we can tell, a first-of-its-kind entry into the domain of CAA tools. Agar embraces the idea that feedback from a human grader is an essential and invaluable part of the learning process. Rather than strive only for complete automation, Agar works to extend the existing usage patterns of graders by emulating red-pen grading as well as division-of-labor methods for large grading tasks. By automating the tasks of score recording and sending feedback to students, Agar works to eliminate the clerical tasks that can otherwise take up so much time in grading. The “write-once” comment system also makes grading in Agar more consistent, and allows graders to increase the quality of feedback to students. Agar is more than just an automated grading system: it is an actual *aid* for assessment, not a replacement for the feedback and expertise that is so critical in the learning process.

VI. Acknowledgments

We would like to thank our loyal and devoted user base for perservering through numerous initial bugs and revisions: instructors Brian Linard, Kristen Miller, Ann Gordon-Ross, and, Dr. Peter Fröhlich and TAs Tim Mauch, Keri Nishimoto, Dave Sheldon, and Ryan Rusich have been invaluable in providing feedback, feature requests, and bug-reports. They suffered the bugs and misfeatures of our early efforts and must also be recognized as a critical part of this project. Finally, we’d like to thank our friendly local system administrator Victor Hill for viewing this as an important project and giving us the necessary support to make it fly.

References

[1] J. B. Hext and J. W. Winings. An automatic grading scheme for simple programming exercises. *Commun. ACM*, 12(5):272–275, 1969.
[2] J. Hollingsworth. Automatic graders for programming classes. *Commun. ACM*, 3(10):528–529, 1960.
[3] A. Patterson, M. l Kölling, and J. Rosenberg. Introducing unit testing with bluej. In *Proceedings of the Information Technology in Computer Science Education Conference*, 2003.
[4] L. Rich, H. Perry, and M. Guzdial. A cs1 course designed to address interests of women. In *SIGCSE 2004 Proceedings*. ACM Press, 2002.
[5] B. T. Tognazzini. *Tog on Interface*. Addison-Wesley, 1996.
[6] U. von Matt. Kassandra: the automatic grading system. *SIGCUE Outlook*, 22(1):26–40, 1994.
[7] T. Winters and T. Payne. What do students know? In *ICER*, 2005.