User's Guide for `autograde` (version 2.0)

Michael H. Goldwasser
`mhg@cs.luc.edu`

22 March 2002

Contents

1 Introduction

`autograde` is a Perl script designed to automate the execution and testing of student programs, as part of the grading process. The script is not in any way intended to provide the subjective evaluation which goes into the grading of student programming assignments. Nor is it intended to give students *instantaneous* feedback on their program correctness. Rather, it is a tool to automate the batch process of compiling, executing and evaluating correctness of students' programs. A battery of tests can be provided by the instructor or by the class as a whole, as discussed in the paper "A gimmick to integrate software testing throughout the curriculum," which appears in the *Proceedings of the 33rd Annual SIGCSE Technical Symposium on Computer Science Education*, Feb. 27 – Mar. 3, 2002, pp. 271–275.

The tool's flexible design was intended for maximal use with a variety of programming assignments. The script is independent of the programming language used for assignments, and an executable to be tested can be built from a single source code file submitted by a student or from multiple source files provided by a combination of the student and instructor. The current version of the script has been developed for a Unix/Linux operating system. Our hope is that instructors will find it easy to retrofit most existing assignments for use with this software. Further discussion on assignment design is given in Section 6.

2 Overview of Process

The script is designed to take source code submitted by students, compile that code, and to run the executable on a sequence of tests. The tests can be specified in one of two ways:

- In the default mode, the sequence of tests is specified by the instructor. Each individual test is described as a command-line statement to be executed in the default shell.
- In a special "head-to-head" mode, each student is expected to submit source code as well as a test plan, specified as a text file with a predefined file format. In this mode, each student's executable is run on each student's submitted test plan. For each student, the following metrics are eventually reported:
  - A student's code is rated based on the percentage of submitted test files on which it executed successfully.
  - A student's test file is rated by considering the number of submitted programs with a demonstrated flaw on <u>this</u> test, versus the overall number of submitted programs with a demonstrated flaw on <u>at least one</u> submitted test.

The "correctness" of a student's program on a particular test is judged by a direct comparison between the output of the student's program and the output of a model implementation, provided by the instructor. An outline of the life-cycle for a typical assignment is shown in Figure 1. Issues related to the design of assignments are discussed in Section 6. Preparing a configuration file and the initial directory hierarchy is discussed in Sections 3.1–3.2. The portion which involves use of `autograde` can be thought of as a single batch process, though Section 5 will discuss situations in which there is advantage in running the distinct stages independently.

Before the assignment is given to the students:

- Assignment description is developed.
- If doing head-to-head competition, test file format must be declared.
- Optionally, instructor-provided components can be given to students, to be linked with each student's own source code.

Before the automated grading process begins:

- Source code must be developed which constitutes a "model" implementation.
- In some cases, additional software components, to be linked with student submissions, are developed by the instructor to facilitate the automated grading.
- Students' source code, and test files if doing head-to-head competition, must be submitted into a separate subdirectory for each student.
- `autograde.config` file must be created for the assignment, specifying information such as, compilation syntax, filenames, sequence of tests, etc. (see Section 3.2)

Distinct stages of grading with `autograde`:

- students' tests gathered from submit directories, if head-to-head competition.
- model implementation must be executed to create answer key for each test.
- For each student:
    - ◦ Copy submitted files to a working directory (if not already present).
    - ◦ Compile the submitted files (perhaps linked with instructor's files).
    - ◦ For each test:
        - — Execute the program, saving output to a file.
        - — Compare the saved output to the model output, saving differences to a file, if not a match.
    - ◦ Generate summary file for this student.
- If head-to-head competition was performed, generate composite metrics.

Figure 1: Assignment Life-cycle

3 Software Usage

3.1 Directory Hierarchy

A separate directory for grading must be devoted to each assignment. This is the base directory from which the `autograde` script must be executed. By default, the hierarchy must contain the following at the onset:

- A subdirectory, `submit`, which contains further subdirecories for each student in the class. Those subdirectories should contain all relevant files as originally submitted by the student.

- A subdirectory, `model`, which contains a single subdirectory containing source code which comprises a model implementation for the assignment.

- A file, `autograde.config`, as discussed in Section 3.2.

In addition, if the student programs are to be linked with any components provided by the instructor, a subdirectory, `ourfiles`, should contain object code for those components.

Beyond these initial contents, the autograde script will create and make use of several additional subdirectories as the process unfolds. First we will discuss the default setting, where tests are provided by the instructor.

- `answers` – a subdirectory containing the answer key for each test. These files are created by capturing the output of the model implementation on each such test.

- `working` – a subdirectory containing further subdirectories for each student, to be used as the working directory for compiling and executing the students' programs. Initially, the source code submitted by the student is *copied* to this working directory and combined with instructor-provided files. Then the source code is compiled, and the tests are executed. An additional subdirectory, `testing`, will be created in each working directory. For each test, the captured output will be saved as a file in the testing subdirectory, as well as a summary of differences between the captured output and the answer key for each test. Finally, inside each working directory, a report providing a summary of the process for the student will be created, as discussed in Section 4.

An important subtlety in the implementation for the working directories is that the student's source code is copied from the submit directory to the working directory when the working directory is first created. Furthermore, the instructor might choose to edit this copy and rerun autograde, while leaving the original version unchanged in the submit directory. The advantage of this option is discussed more fully in Section 5.

When running in head-to-head mode, an additional subdirectory, `competition`, is created to handle all facets of the competition. Within that subdirectory, the following are created:

- `input` – the test files submitted by the students are gathered from the original submit directory and placed here.

- `answers` – this directory contains the answer keys for the student-provided tests (as opposed to the answer keys for instructor provided tests).

- `working` – this directory serves a similar purpose as did the working directory in the default mode, however it should be noted that the working directories for the head-to-head competition are maintained separately from the original working directories.

- `results` – for each student, a succinct report is created summarizing the successes and failures of that student's code. This report is used at the end, to compile overall metrics for the competition, and it is different from the more verbose report created in the working directory.

3.2 The `autograde.config` File

For any given programming assignment, a configuration file must be created to customize the process for the particular assignment. The file will be interpretted, using Perl syntax, from within the execution of the `autograde` script. A sample of a configuration file is given in Figure 2. Within the configuration file, any combination of the following variables can be optionally defined:

- Regarding compilation: the compilation of student programs can be controlled as follows. An individual compile command can be specified for each submitted source file. After these individual commands have been executed, a final compile command can be specified. The variables involved are:

  - `$sourcesuffix` – any file with a name that ends with this string is considered to be source code.
  - `$compile_file` – this string specifies a single shell command which is to be executed once, for each file classified as source code above. All occurrences of the substring `FILE` in the string will be replaced with the actually source code filename.
  - `$compile_final` – this string is a single shell command which will be executed once, after the individual compile commands are completed.

  Depending on the programming language being used, one or both of the compilation techniques can be used. For example, if using C++ it may be necessary to compile each source code into object code, followed by a final linking step. Alternatively, the entire compile and linking can be delayed into a single step through `$compile_final`, perhaps by use of a Makefile. Alternatively, if using Java, no linking is necessary and so compiling each individual file is sufficient.

- Regarding instructor-provided files:

  - `@our_files` – this is an array of strings, designating the file names of any files which are provided by the instructor but needed as part of execution. The specified files are assumed to be precompiled, and must exist in the `ourfiles` subdirectory. A link is created to each such file, from within the working directory for each student.
  - `@excluded_files` – this is an array of strings, designating particular file names which should be ignored if found in a student's submit directory. Specifically, if using instructor-provided files, it is probably useful to make sure that those versions are used, as opposed to any related files submitted by the student.

- Regarding instructor-defined tests:

  - `@tests` – this is an array of strings, where each string designates a single test as a shell command to be executed from within a student's working directory. This command will be executed, with the resulting output saved to a file. That file will then be compared to the answer key produced by the model implementation.

  - `@samples` – this is an array of strings, using the same format as was used for `@tests`. The difference in treatement is that samples are used to capture output of a student execution for later examination, yet for which no direct comparison to the model implementation need be made.

- Regarding students' submitted tests: If head-to-head competition is used, the following two variables must be properly defined.

  - `@HTHinput` – this is used to detect which of a student's submitted files is the one to be interpretted as the test file for competition. The precise rule used is that `@HTHinput` is an array of strings, and the test file is expected to have a file name which contains one of those strings as a case-insensitive substring.

    This rule was developed to handle the fact that if students are asked to submit a test file named "`inputfile`" some will invariably submit files with names such as "Inputfile," "input file," "inputfile.txt," "myinput" and so on. If more than one submitted file matches the rule, an arbitrary match might be selected.

  - `$HTHcommand` – this string specifies a single shell command which is to be executed in the working directory for each student, once for each test file in the competition. All occurrences of the substring `FILE` in the string will be replaced with the actually filename of a test input.

- Regarding generated report cosmetics: Section 4 discusses the reports generated by the script. Several variables effect the creation and cosmetics of these reports, which may be helpful if those reports will be given back to the student.

  - `$asgn_name` – A string identifying the assignment

  - `@required_files` – An array of strings declaring precise file names, each of which was expected to be submitted as part of the assignment. For any name in this array, but not submitted, an additional line will be added to the generated report declaring this omission.

  - `@readme` – An array of strings used to detect a single readme file, when one is expected as part of the assignment. The syntax for this array is the same as was used with `@HTHinput`. (again, to recognize variations such as "Readme.txt" or "read_me"). If no match is found, an additional line will be added to the generated report declaring this omission.

  - `$extracreditstart` – This numeric variable specifies the index into the array `@tests` at which point the tests begin relying on functionality which was deemed as extra credit in the assignment. By default, the tests that were specified in array `@tests` are labeled as *test1*, *test2*, *test3*, etc. The effect of setting this variable is to relabel the final portion of those tests as *extra1*, *extra2*, etc. Please note that the indicies start at 1, therefore if the array `@tests` contains 10 entries, and the variable `$extracreditstart` is set to 7, then the tests will be labeled as,
    test1, . . . , *test6*, *extra1*, . . . , *extra4*.

```
$sourcesuffix = ".java";
$compile_file = "javac FILE";
$compile_final  = "";

$asgn_name = "Prog 4 -- Hand";
@required_files = ("Hand.java");
@readme = ("read");

@our_files = ("Card.class","CardsDriver.class","CardsGrader.class",
        "List.class","ListImplementation.class","Position.class");
@excluded_files = ("Card.java","CardsDriver.java","CardsGrader.java",
        "List.java","ListImplementation.java","Position.java");

@tests = ("(java CardsGrader Y ../../input/t1)",
          "(java CardsGrader Y ../../input/t2)",
          "(java CardsGrader Y ../../input/t3)",
          "(java CardsGrader Y ../../input/t4)",
          "(java CardsGrader Y ../../input/t5)",
          "(java CardsGrader Y ../../input/t6)");
$extracreditstart = 5;

@HTHinput = ("input","test");
$HTHcommand = "(cat ../../../input/quit | java CardsGrader N FILE 100)";
```

Figure 2: Sample file `autograde.config`

3.3 Command Line Arguments

By default, the `autograde` script does nothing, if executed without any command-line arguments. The following command line arguments are accepted:

- Choice of mode:

  -H
  By default, the script uses tests designated by the instructor in `autograde.config`. If this flag is declared, then the script uses tests designated by the students through the head-to-head competition mode. In this case, all work will be done within the `competition` subdirectory.

  Note: If the desire is to perform instructor's tests as well as the competition, `autograde` must be exectuted separately in each mode.

- Declaring subset of phases (as shown in Figure 1) to complete:

  -g
  This flag is only meaningful if in head-to-head mode. In that case, the script will search students' submit directories in an attempt to gather all of the test files.

  -m
  The script will compile the model implementation and then execute it on all applicable tests, saving the output to form answer keys.

-p
The script will copy and compile each student's source code in a working directory (but will not execute any tests).

-e
The script will copy, compile and execute each student's submission, for all of the desired tests.

-s
This flag is only meaningful if in head-to-head mode. In that case, the script will calculate the final competition metrics for each student based on the completed tests. A report of these metrics is written to standard output.

-a
Do all relevant phases. If in standard mode, this is equivalent to the '-m -e' flags; if in head-to-head mode, this is equivalent to the '-g -m -e -s' flags.

- Other effects on process:

  -l <u>students</u>
  By default `autograde` processes all students found in the submit directory. If a comma-separated list of students is declared in such an argument, only those students will be processed by the script.

  -t <u>sec</u>
  By default, student program executions are timed out after 10 seconds. This argument allows that default to be changed.

  -r <u>prob</u>
  When executing in competition mode, the default behavior is that each program is run on each test. If a class is so large that the computation time becomes restrictive, it is possible to rely on random sampling. If this parameter is specified, each given program/test combination will be executed independently with the specified probability.

  -h
  Displays usage information for script and list of accepted arguments.

  -d
  Runs script in debug mode. While processing, more information is sent to standard output then when running in the default mode.

  -q
  Runs script in quiet mode. While processing, less information is sent to standard output then when running in the default mode.

- Modifying default hierarchy: the remaining arguments can be used to change the default names used for the directory hierarchy.

-F	<u>config</u>	Configuration file	(Default: "autograde.config")
-S	<u>dir</u>	directory of student submissions	(Default: "submit")

-O dir	directory containing "our" files	(Default: "ourfiles")
-M dir	source for model solution program	(Default: "model")
-C dir	competition subdirectory	(Default: "competition")
-W dir	original working directory	(Default: "working")
-A dir	directory containing model answers	(Default: "answers")
-I dir	directory of test input files	(Default: "input")
-R dir	comma separated list of directories with competition results	(Default: "results")

4 Reports Generated

For each student, with given `userid`, a report which summarizes the proccessing will be created as `working/userid/userid.autograde`. The report contains a header, and then contains additional information based on the particular steps of the grading process which were attempted. Its format is intended to be helpful to the grader, and appropriate for returning to the student. Information which may be included in this report is:

- When searching the submit directories, only top-level files are considered. If any subdirectories exist, they are not checked, however a note is added to the report about the existance of those subdirectories.

- If gathering test input files from student directories, a warning will be printed if no such file was submitted.

- For any other "required" files which cannot be found, a warning will be printed.

- For each step of attempted compilation, a note will be placed either confirming successful compilation, or else noting failure and including the output of the failed compilation.

- For each test which is exectuted, a line will be included in the report designating the name, outcome, and length of time in seconds of each test. The outcome might be labeled as "succeeded," "failed," "Timed Out," or "completed" in the case of a sample or the model.

When in head-to-head mode, two reports are created for each student. One report, exactly as described above, is created as `competition/working/userid/userid.autograde`. A second report is created and copied to `competition/results/userid`, and used later for compling overall metrics for the head-to-head competition. This report contains only a distilled summary of the test outcomes.

5 A Typical Grading Cycle

When run with the "-a" flag, the `autograde` script executes all phases of the grading process, as shown in Figure 1, as a single batch process. In an ideal world, this may be all that is needed. In practice, however, there are many situation in which distinct stages must be run separately. We first discuss the standard mode, in which programs are submitted by students, yet tests are provided by the instructor. Then we will discuss the head-to-head competition mode.

5.1 Standard Mode

In this setting, the full process consists only of two phases, the creation of the model solutions, and then the processing of the student submissions. That is, running

```
autograde -a
```

is equivalent to the combination

```
autograde -m
autograde -e
```

Since the tests and the model implementation are provided by the instructor, the answer keys could be created with the "-m" flag, even before students submit their own implementations. In reality it is almost always advisable to create and examine the answer keys before using them as the basis for checking correctness of student implementations.

Once the answer keys exist, student implementations can be executed with the "-e" flag. However, before taking the time to do all executions, it is often informative to simply test whether required files were indeed submitted, and if so, whether they compile successfully. Running the script with the "-p" flag will cause it to copy and compile (but not execute) submitted files, creating a report which summarizes any problems.

When ready to execute all the tests, running with the "-e" flag by default processes all students. The "-l" flag can be used to select a subset of the students, specified as a comma-separated list of userids, for example as

```
autograde -e -l alice,bob
```

The main advantage offered by this option is the ability to process late submissions received after the initial grading, or to reprocess implementations after the instructor has modified the original source code. Whether or not to modify a student's code is up to the instructor's discretion, but there are some situation in which it seems appropriate or useful.

- As an extreme example, there are cases in which a student finally gets the program working, adds some comments just before submitting, and ends up submitting source code which causes a compilation error. Whether or not a grader chooses to penalize such a student, it is probably worthwhile to fix the compilation error and see how the modified code executes on the tests.

- There are cases where a student's "correct" implementation will fail the automated tests because the correctness is judged by comparing the standard output of the program versus a model implementation. The assignment design can try to minimize the likelihood of such cases (see Section 6), but individual cases may slip through the cracks. For example, a student implementation may include debugging output which should have been removed before submission, yet remains. This spurious output would cause the implementation to fail the tests en masse.

- Finally, after doing an initial evaluaton of the correctness of students' original code, it is common, as a grader, to debug the code and suggest changes which would improve the implementation. After recording the original performance report, modifications can be made to the student's code, and then the debugged code can be retested for correctness. This often allow's the grader to determine whether an obvious and correctable mistake was the sole mistake, or whether further problems exist.

In order to support such modifications, the software uses the following rule. A student's submitted source code is assumed to be in a submit directory. That code is originally copied by the script into a separate working directory at which point it is compiled and executed. The instructor is free to edit the version of the code in the working directory, while leaving the original version in the submit directory. In this case, further applications of the script will recompile the version as it appears in the working directory.

5.2 Head-to-Head Mode

When administering a head-to-head competition, even more issues arise. In this setting, running

```
autograde -H -a
```

is equivalent to the combination

```
autograde -H -g
autograde -H -m
autograde -H -e
autograde -H -s
```

Given that the lion's share of time is spent during the all-pairs execution of the student programs, it makes great sense to validate the success of the two earlier phases before beginning those executions. When gathering student tests, two possible pitfalls exist. First, a student might submit a test, but using an unrecognized filename. Secondly, a student might submit a test which for one reason or another, is not parsed properly. In either of these cases, the student is likely to receive a score of zero for the testing portion of the compeition, though this may seem unfair to the student. Human intervention can generally be used at this early stage to avert such problems, though it is up to the grader as to whether or not to spend the time to intervene.

Once the input's are gathered, then the model answers can be constructed, and the heart of the student competition can begin. As in the previous section, there are cases where it may be appropriate for a grader to modify a student's source code before running all of the tests, though in a competition such editing would probably be more restrictive. It should be noted that the 'working' directory within the competition subdirectory is independent of the 'working' directory that was used when run in standard mode. That is, the version of the code in the 'competition/working' directory is originally created as a copy of the original submit directory version, and the grader can choose to edit this copy with discretion.

The situation which is signficantly more complicated in head-to-head mode, is that of processing a late submission. In standard mode, the grading of one student's submission was uneffected by other students' submissions, and thus the '-l' flag could be used to individually process late submissions. In head-to-head mode, processing a late submission involves not only running the implementation on many other tests, but it involves a newly submitted test which must be used on all previously submitted implementations. For demonstration, we consider processing two late submissions from 'alice' and 'bob' after the remainder of the grading has been completed. In this case, the following should be done:

```
autograde -H -e -l alice,bob
autograde -H -g -I lateinput -l alice,bob
autograde -H -m -I lateinput
autograde -H -e -I lateinput -R lateresults
autograde -H -s -R results,lateresults
```

The first line has the effect of executing alice and bob's implementations on all previously submitted tests. The remaining four lines administer testing of all implementations on the two newly submitted tests. When processing student implementations, all tests which are found in the input directory are used. Therefore, the only way to run implementations on two distinct tests is to create a second input directory containing only those tests. Therefore, the second line above gathers the two new inputs, but places them in a new directory titled 'lateinput' in this example. The third line creates model solutions for those two tests, while the fourth line executes all implementations (including alice and bob's) on the two new tests. However, it is important to preserve the result summaries of the many previously executed tests, and therefore the fourth line declares that result files for these tests be placed into a separate directory, titled 'lateresults' in this example. After the first four lines have been executed, all possible implementation/test pairs have been executed, however the result summaries are broken across two directories, 'results' and 'lateresults' in this example. Therefore, when compiling the final statistics for the overall competition, it is important to specify that results be combined from both of those directories.

6 Impact on Assignment Design

`autograde` was created with a goal of being amenable with as many existing programming assignments as possible. In this section, we discuss the flexibility of the tool and its impact on the design of assignments.

6.1 Checking Correctness

The most important issue in assignment design is ensuring that correctness checking can be automated. Assignments must therefore be formally specified so that the notion of "correct" behavior is unambiguous. In its simplest form, the correctness of a student implementation on a given test can be verified by comparing its captured output directly to the output of a model implementation. At face value, such a direct comparison would be sufficient in some settings. Yet there are many settings in which looking for an exact match is tenuous if not completely inappropriate.

- For any assignment, finding a perfect match between a student's generated output and a model solution requires very explicit guidelines from the instructor and great discipline on the part of the student. It may work, but it is likely that if a student makes a mistake in the formatting of the output, that student's implementation will indiscriminately fail all tests.
- For some assignments, the notion of correctness is well-defined yet not unique. For example, one of the assignments distributed with this package involves an aspect of a game of cards. At times, the rules allow for more than one legal play. In such a setting, relying on an exact match between the behavior of a student implementation and that of a model implementation is not a legitimate technique for verifying correctness.
- Some assignments involve students writing components which do not directly produce any textual output. For example, the student's code may be responsible for the creation or maintenance of an internal data structure.

In order to best address the above situations, the design of `autograde` incorporates the two following techniques. First, students' submitted source code can be linked with additional components provided by the instructor to produce executables with a varied behavior. Secondly, rather than directly capturing the output of the program execution, `autograde` captures the standard output produced by a more general shell command, as defined within `autograde.config`. In cases where a direct match of a program's output suffices, this shell command may simply initiate the program execution. But in other cases this design allows the output of the students' executable to be post-processed when piped to standard system tools (e.g., grep, awk) or even custom-design tools. Generally, the combination of one or both of these techniques can be used to develop fully automated correctness checking.

- To avoid the pitfall of student generated output which does not precisely conform to guidelines, output can be generated indirectly through the use of components provided by the instructor. Alternatively, it may be reasonable to compare the students' original output to that of a model solution, so long as system tools are used to pattern match portions of the output, rather than relying on an exact match.
- For assignments in which the notion of correct behavior is not unique, checking can be effective using either of these techniques. For example, the correctness can be check in real-time by linking in an instructor's compoenent which observes the behavior of the student program, raising a red-flag when a behavior flaw is detected. Alternatively, the non-trivial correctness checking can be implemented through a post-process analysis of the original execution output.
- For assignments in which no classical output is generated, non-trivial components can be linked in to verify correctness. For example, a checker can compare an internal data structure produced by a student to a reference structure built by the checker. Meaningful output can be produced in the case of a mismatch.

6.2 Test Set Format

In spirit, test sets could be specified either implicitly through software components which interact with student submission while performing a battery of tests, or explicitly as a file using a defined format. When testing is to be specified by the instructor, `autorade` can be configured to use either of these means through the `@tests` parameter in `autograde.config`.

For a head-to-head competition, students must submit their tests as a text file which strictly adheres to a file format defined by the instructor in the assignment guidelines. The script will search for this input file in the students submit directory, and the file will be passed to other students' executables either redirected as standard input or through the use of a command-line argument. This choice is left to the configuration of autograde, through the `$HTHcommand` parameter of `autograde.config`.

It is imperative that students rigorously follow the prescribed standards. Potential pitfalls for a student include submitting test input that is essentially ignored for not adhering to the standards, as well as submitting an implementation that does not properly parse the standard input format. In either such case, the student risks failing the competition en masse. In fact, there is a compound risk that the student will submit a flawed input format that is self-consistent with a flawed parser. Unless the intent of an assignment is specifically geared towards a student's ability to parse or error-check input, the overwhelming conclusion is that the instructor should provide all students with a robust front-end parser so as to avoid most such tragedies.

Admittedly, a traditional text-based "console" interface may indeed be unappealing to modern students. We wish to point out that in a modular setting, the instructor can provide both a textual interface as well as a separate graphical user interface. Students could use the GUI for most of their development if they wish, but would eventually need to use the textual driver for developing their test file. As a side effect, it is advantageous for the student to see both such interfaces. Even though the GUI might seem attractive for the end user, the textual interface is quite valuable during a debugging phase on large tests. It is quite time consuming to repeatedly step through the first 40 mouse-clicks to reach the bug on the 41^{st}, yet easy if such a chain of commands can be read from a file.

6.3 Test Set Size

As the head-to-head competition is structured, a student receives credit if her test set exposes one or more flaws in another student's implementation. Thus students will gain an advantage by developing larger and larger tests. For this reason, a strict limit should be placed on the inherent size of the tests. The form of this limit will depends on the particular assignment and its enforcement can be enacted through a front-end driver or preprocessing.

6.4 Sample Assignments

In order to demonstrate some of these issues, the distribution package includes two sample assignments, complete with all relevant files.

7 Technical Issues

7.1 Portability

This script has been designed and tested using Perl5 under both Solaris and Linux. We have not made any attempt to port this to a Windows environment, though perhaps it is possible. We invite comment on any user's experience in regard to porting.

The most technical issue, in regard to portability, is the ability to execute student programs while setting an alarm as a timeout, and subsequently killing any child processes upon timeout. As much as possible, we have used existing Perl commands hoping to rely on its portability. However, we have relied on standard system calls (cp, pwd, date, diff, ps) in places. Additionally, we rely on the ability to redirect or capture the standard output and standard error.

7.2 Security

No effort has been made to safeguard against malicious students. During the execution of this script, a student's program will be executed as a process using the effective user ID of the grader. This presents a huge security whole in terms of the file system and machine on which grading is done, as the student's programs might interact with the operating system.

There are many common techniques to safeguard against such problems; they simply have not been implemented in the current version of `autograde`. Possible safeguards include running student programs with the root directory remapped to the working directory, and creating a separate userid with limited system permissions, for the purpose of grading.

8 Download

The software, as well as a demonstration of its use through several sample programming assignments, can be found at `http://www.cs.luc.edu/~mhg/autograde`.

9 Feedback

Please feel free to contact the author via email to discuss any experiences you have, either positive or negative. If reporting a bug, please give specific information about your system and include additional files, if relevant. If further updates are made to the software or the documentation, it will be distributed through the website referenced in Section 8.