A Secure Marking System
Problem writers guide

Bruce Merry and Carl Hultquist

January 24, 2014

1 Introduction

This document is aimed at problem authors who wish to put their problems into the secure marking system. It does not cover the setup or use of the marking system itself — for that, refer to the administrators guide.

The marking system has a number of important features and restrictions:

- Marking is completely automatic and based only on the correctness of the output. For this reason, solutions that fail to compile, fail to produce output, crash etc., will score zero marks, even if they are very nearly right. The system has no provision for marks to be modified; any manually assignment must be handled elsewhere, such as in a spreadsheet.
- There are three types of problems:

  run The program is expected to read some input data, perform some computations, and produce some output data. The input data is kept secret from the contestants.

  data No program is handed in. Contestants are given the input data via the web-based system. They hand in the corresponding output file.

  interactive The contestant's program interacts with an opponent via pipes (stdin and stdout).
- There is exactly one input source and one output sink for a problem run. Although there can be multiple test cases, no test case can consist of more than one input or output file.

2 Generating a template

The script `scripts/problem-template` can be used to generate an initial problem directory. Run it without parameters to see how it works. It also generates a skeleton evaluator. The Makefile is set up to compile the sample solution and evaluator from the same source file. The program determines its use from the number of command line parameters.

If the output from a problem is unique, the generator may be asked to create a diff-based evaluator. This will handle all details of evaluation, and only input data and a model solution are required. The language specified to the template generator determines the language that the model solution is expected to have.

Unlike other scripts, this script should be run from the `problems` subdirectory. This is to allow it to be used standalone i.e. independently of an evaluation tree.

There is also a web interface to the template generator, in the `misc` subdirectory of the SMS tree (note that this is not linked into a contest tree; you have to go to the original source). Copy or link `templategen.html` and `templategen.php` into a directory in the web space and point people to `templategen.html`

FIXME: integrate the front-end into the system.

3 Files

Most of the required files are generated by a template generator. You will just need to fill them in. The contents are documented here, but in most cases the comments in the files themselves are adequate.

A problem consists of the following files:

- An options file, that holds metadata for the problem
- A problem statement
- A combined evaluator and sample solution
- A Makefile for the evaluator
- Input files

The format of these files is described in the following sections.

3.1 `options` file

This file is written in the language of the configuration engine, which is described in detail in the developers guide. This is a powerful syntax, but problem authors will generally need only the following:

- `setting` — enables a boolean setting
- `setting='value'` — sets a value
- `setting=number` — sets a numeric value

Single quotes can be included in values by escaping them with a backslash. Blank lines and comments preceded by a `#` are ignored. Numbers can be suffixed with `k`, `m` or `g` to indicate kibibytes, mebibytes or gibibytes.

Although any setting can be placed in this file, the settings relevant to problem authors are

`problem.type` One of `run`, `data` and `interactive`, specifying the type of the problem.

`problem.author` The name of the problem author.

`problem.runs` The number of test cases.

`problem.feedback.runs` If set, enables detailed feedback for the problem. If set to a number N, the first N test cases are used, otherwise all are used for detailed feedback.

`problem.score.sample` The maximum number of points for the sample test case. This is used to sanity check handins.

`problem.score.real` The maximum number of points for the real test cases. This is used to sanity check model answers.

`problem.sample` The name of the source file for the evaluator. This setting is not currently used, but should be provided to future-proof your problem.

`problem.max_time` The maximum time for which a solution may run, in seconds.

`problem.max_memory` The maximum total memory that a solution may use.

`problem.max_stack` The maximum stack space that a solution may use.

`problem.max_file` The maximum size of the output files that a solution may produce.

`problem.compile.max_time`, `problem.compile.max_memory`, `problem.compile.max_file` Limits on compilation of a solution, rather than on the solution itself.

`problem.stdin`, `problem.stdout` Boolean settings; if provided, they indicate that solutions to *run* problems will read data from standard input and write to standard output respectively.

`problem.input_extension` If set, overrides the default extension of `.in` for input files (the dot should be included in the value).

`problem.output_extension` If set, overrides the default extension of `.out` for output files (the dot should be included in the value).

`problem.reveal_input` By default, the contestants' programs in interactive tasks cannot access the input data (the evaluator should provide any data). This allows hidden information to be stored in the input files. If this flag is set, then the input data will be made available. This is not recommended and the option exists largely for compatibility with old problems that require it.

`problem.format_check` If set, and if the problem is a data problem, the handin system will check that submissions are correctly formatted by running the evaluator with an extra `--format-check` argument. To indicate that the submission should not be accepted, the evaluator should give a zero score, with a reason starting with `Invalid`. The evaluator should be careful not to leak any information about the test data or the submission's correctness when run with `--format-check`.

It is recommended that `problem.max_time` is set, but that the other resource limits are omitted and set on a per-contest basis.

3.2 `data` directory

This subdirectory of the problem directory contains the input data. The test cases are numbered from 0 to `problem.runs`. For test cases with a single test run the filenames are of the form *problemN*`.in`, where *problem* is the short name of the problem and *N* is the test case number. For test cases with multiple test runs, the runs are numbered using the lower-case letters a..z, and the filenames are of the form *problemNC*`.in` where *C* is the test run number. Note that the extension here is always `.in`, independent of the setting of `problem.input_extension`. Test case 0 is the sample input (placed in the problem description by LaTeX, and used by the web front-end for sanity checking).

In addition, your evaluation program may use per-testcase data to make the evaluation process easier. These should be named the same as the input data but with the prefix `e`. The obvious use is for problems where there is only one correct answer, in which case `diff` can be used for comparison. The template generator has a an option to automatically generate an evaluator that does this, and automatically generates these files from the input data using the model solution. Another possible use is for relative evaluation tasks (where the score is based on the best solution of everyone else), but this is not completely supported without hacks.

For interactive tasks, the files in this directory are fed to the opponent. However, they are not made available to the client, so interactive tasks must provide for the opponent to send any initial state to the client.

3.3 `src` directory

This subdirectory contains the source code for the evaluation program and a `Makefile` to compile it. The compiled evaluation program should be called `e`*problem*[1]. Although the source file is not

[1] It must be a free-standing executable; it is possible to get a Java program working with some dirty hacks, but it is not supported.

currently used for anything, it is recommended that to future-proof your program you should ensure that

- your solution is called *problem*`.`*ext*, where *ext* is the suitable extension for your language;
- the compiled program acts as a sample solution when given no arguments.

In future, these files may be automatically found and used to check that a model solution scores 100%.

Where the template generator is used to generate a diff-based evaluator, the generator provides the evaluator, and you need only write a sample solution.

It doesn't matter what other files are in this directory, as long as the default Makefile rule will compile the evaluator. The evaluation system will pass the Makefile variables to indicate the compiler and interpreter flags used by the system itself, so that the evaluation program can be built in the same way. For example, `CXX` will be set to the C++ compiler (as set by `compiler.gcc.compiler`, or the default value). The recommended use is to set `CXX` to a default value in the Makefile, so that compilation will succeed even if `make` is run manually. It is also recommended that you do *not* use the flags variable, as it is usually set for optimisation rather than error checking.

See section 4 for the interface the evaluation program should use to communicate with the rest of the evaluation system.

3.4 Problem description

If one wishes to make problem descriptions available through the web interface, there are five formats supported. The description must be named *problem*`.`*extension*. The supported extensions and corresponding file formats are

`.txt` Plain text.

`.html` HTML — however, provide only the contents of the body, not a complete HTML file, as the contents will be embedded in a larger document.

`.pdf` Adobe Portable Document Format

`.doc` Microsoft Word document. This is strongly discouraged, as Word is a proprietry format that cannot be read on all machines.

`.tex` LaTeX. The system provides special support for typesetting in LaTeX (see the next section for details). The output is a PDF file.

3.5 Typesetting problems with LaTeX

Preparing the problem text separately from the problem data is error-prone, as there are multiple versions of some data (e.g. time limit, sample input, sample output etc.) If there is a summary page, the chance of error is even greater. To reduce the risk, the system provides a LaTeX class for typesetting problems, and support to embed the actual problem data in the problem text.

The commands are best illustrated by an example:

```
\documentclass{saco}
\begin{document}
\maketitle
\section{Introduction}
Introductory stuff.
\section{Task}
Specific stuff about the task.
\section{Example}
```

```
A specific example.
\inputformat
Describe the input format.
\sampleinput
\outputformat
Describe the output format.
\sampleoutput
\section{Constraints}
Constraints on the data.
\timelimit
\section{Scoring}
The scoring scheme.
\end{document}
```

You should not include any preamble commands (like `\usepackage`), because when the combined document is created, this file is included in the body of the bigger document with the header and footer stripped off.

You can compile this directly with `pdflatex` (as long as the class file is in the TEX path), but none of the fields from the problem will be filled in. The recommended approach is to use the web interface, and select the problems from the problems menu.

For both the *data* and *run* problem types, it is expected that the evaluator will work like a sample solution when given no arguments; this is used to generate the sample output (which is done automatically when using a template; see the next section). For *data* problems it may be necessary to add code that will handle the sample input, if there is no model solution code.

4 Evaluator interface

For each problem there must be an evaluation program (whose location is described in section 3.3). It should read the input data, the output data, and optionally the precomputed per-testcase input file mentioned in section 3.2. The paths to these are given on the command line in this order. There will always be a third parameter but if no precomputed files were provided then this parameter will be meaningless.

An evaluation report should be produced on the standard output. The report has no fixed format except for the last two lines and should be a detailed log of everything the evaluation program is doing. The last two lines must look something like this:

```
Score:  5
Reason: Partially correct answer (5 of 10 items correct)
```

Note that there are two spaces in the first list, so that the data in the fields is aligned.

There are no requirements on what the maximum score for a test case may be; it may even vary from one test case to the next (this is another possible use of the per-testcase evaluation data). The evaluation system will simply total up all the scores. Unless you wish to unevenly weight the problems it is suggested that the maximum total score summed across all the test cases is 100. The explanation should be short yet descriptive as it is used to construct the summary report.

Below is a sample evaluation output:

```
Reading input data...
Reading contestant's output file...
Contestant found 46 rectangles
Verifying contestant rectangles...
Contestant rectangles are all valid
Solving problem...
There are 92 possible rectangles.
```

```
Score:  5
Reason: Contestant found 46/92 rectangles
```

If the evaluation program encounters a condition it did not expect it should exit with a non-zero exit code. This will cause the evaluation system to halt immediately so that manual intervention can be taken. The C language function `assert` is a good way to do this.

The evaluation program should be robust and able to handle any possible output file including incorrect formatting, binary data, ASCII NUL characters and so on. However it may safely assume that the output file does exist.

The case of an interactive task is slightly different. The evaluation program is used as the opponent for the user's program. The standard output is connected to the user's stdin and stdin is connected to the user's stdout. The two parameters are the input file to use and the output log to write. The output is treated as a normal evaluation log as described above. For interactive tasks it is important that the evaluation program is robust since there are fewer safety checks in place. In addition the web handin system currently allows the contestant to upload test input, so the evaluator must be extremely careful to check the input to avoid crashing or even compromising security. If the test input is invalid this should be reported to the logfile and the evaluator should exit before producing any output to the client. If different strategies are to be used for the secret and user test data, the secret test data should include a magic number to identify itself to the evaluator.