Olympiads in Informatics, 2009, Vol. 3, 60–66

Moe – Design of a Modular Grading System

Martin MAREŠ

Department of Applied Mathematics, Faculty of Mathematics and Physics, Charles University
Malostranské nám. 25, 118 00 Praha 1, Czech Republic
e-mail: mares@kam.mff.cuni.cz

Abstract. Programming contests often employ automatic grading of submitted solutions, but frequently in an ad-hoc way. This article describes our attempt at creating a modular and flexible grading system called *Moe*, which is not tied to the specifics of a single contest.

Key words: automatic grading, Moe, mo-eval.

1. Introduction

Many programming contests in the world, including the IOI, employ automatic grading of the contestants' solutions. This is accomplished by running them on batches of input data and testing correctness of the output. Time and memory limits are usually enforced during the process, which allows to take the efficiency of the algorithm into account.

During the last two decades, a multitude of such evaluation systems have been developed, but most of them are tied to the specifics of a single contest and they are usually neither publicly available nor well documented. This leads to waste of effort by re-implementing the same things over and over, and also to repeating mistakes somebody else has already made and understood.

This variability can be observed even within a single contest. Most of the IOI host countries have used grading systems developed for their national contests, often extensively modified to handle the different conditions of the IOI. Similarly, the regionals of the ACM ICPC differ in their contest environment between regions.

It was repeatedly suggested that a single unified contest system should be built and used at all the major contests. However, it has been argued that the differences between contests would make the system too complicated and cumbersome to maintain. Also, diversity and the ease of experimentation with new ideas are too valuable to lose.

In our previous paper (Mareš, 2007), we have proposed a modular grading system instead, that is a set of simple, yet flexible modules with well defined roles and interfaces. A contest organizer would then pick a subset of the modules and use them as building blocks of his contest environment together with other locally developed parts. This allows us to minimize the effort without sacrificing flexibility.

This paper is a report on the state of development of our modular system called *Moe*.

2. The Design of Moe

Moe is a successor of our previous system (MO-Eval), originally developed for the contests we organize, and subsequently generalized. Its primary target is Linux, but most modules should run on any POSIX-compliant operating system. The sole notable exception is the sandbox, which is intimately tied to the details of the OS and of the CPU architecture.

The source code of Moe and the (so far incomplete) documentation are available under the terms of the GNU General Public License from the website mentioned in the references.

2.1. *Available Modules*

Moe currently contains the following modules:

- *sandbox* – runs the contestant's solution in a controlled and secure environment, limiting its execution time, memory consumption and system calls. It is the most mature part of Moe, already in use at several contests (see below for the list of applications). The current version of the sandbox requires a recent Linux kernel on the i386 architecture. A port to the amd64 architecture is near completion, but it requires fixing several security issues in the kernel ptrace interface first, as noted by Evans (2009).
- *judges* – a set of utilities for comparing the solution's output with the correct answer at the given level of strictness, which can range from ignoring white-space characters to ignoring the order of lines or all tokens. The judges are built upon a library of functions for strict and fast parsing of text files, which can be easily used as a basis for custom judge programs. This part is also mature.
- *evaluator* (also known as *grader*) – this module controls the whole grading process. It calls the compilers, the sandbox and the judges as described by configuration files. It handles multiple types of tasks: e.g., batch, interactive, open-data. We have a reliable implementation in Bourne shell, but it is unnecessarily hard to maintain, so we plan to rewrite it in a higher-level language (most likely Perl or Python) in near future.
- *queue manager* – since the evaluation of tasks at a big contest must be performed in parallel, the queue manager can be used to maintain a queue of solutions and distribute them among graders running on multiple machines. Finished, but needs lots of polish to make it usable by a wider audience.
- *submitter* – handles submitting of solutions by contestants and passing them to the evaluation system. While the submitter has been designed for competitions which do not use a web-based interface, it can be also used as a clean interface between the web modules and the rest of the system. Working, but needs revision.

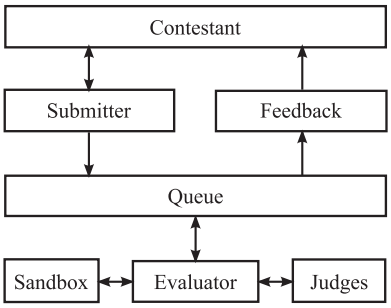

Fig. 1. Typical interconnection of modules.

- *test suite* – the correctness and security of most modules is critical to the success of the contest. We are building a test suite which tries to cover all known edge cases and attacks on system security. The test cases contain unit tests for various functions, regression tests for historic bugs, and security tests inspired by the analysis of possible attacks by Forišek (2006).

In near future, we plan to add several new modules:

- *feedback* – processing of evaluation results and generation of various reports (e.g., score tables, sheets with detailed feedback for contestants).
- *scoring* – multiple small modules for advanced scoring strategies, like gradual time limits (programs near the time limit get partial score) and approximation tasks (points are awarded depending on the precision of the result).
- *supervisor* – controlling several hundred computers at a big contest is no easy task. This module will maintain a queue of jobs (usually snippets of shell scripts or request to distribute files) and schedule their parallel execution at given machines, possibly using a tree-like topology to spread the load.

2.2. *Programming Languages*

Most parts of the system are free of assumptions on the programming languages used in the competition. The only dependencies are in the evaluator and in the sandbox. The evaluator needs to know how to compile and run the programs, which can be of course easily configured. The sandbox has to be set up to permit the system calls issued by the language's runtime libraries.

Pascal (compiled by either GPC or FPC), C and C++ (compiled by GCC) are supported since the first release of Moe and there are no known problems with them, except for the occasional slowness of Pascal I/O libraries.

C# compiled by Mono turned out to be more problematic, because its runtime libraries use various unusual system calls, which are forbidden in the default settings of the sandbox. Also, Mono spawns multiple threads even for trivial programs. We have patched the runtime environment to avoid threading and the most problematic system calls, while the rest is handled by a language-specific configuration of the sandbox.

We have added experimental support for Haskell recently, compiled by GHC.

Adding further compiled languages should be easy, as long as their runtime environment behaves in a sane way. Interpreted languages are also supported, but except for interpreting the C# byte code, they did not receive much testing yet.

2.3. *Interfaces*

To facilitate interconnection of the modules, each of them is written as a stand-alone program with basic parameters passed as its command-line arguments. In addition to this, we accompany every solution by a *status file*, which collects all information related to grading of the solution. All modules can contribute their bits: the submitter records which programming language has been used, the sandbox reports the execution profile (time, memory, number of system calls), the evaluator informs about the results of each test case, while the queue manager annotates on which grading machine has served the task and how long did the task wait in the queue.

We have resisted the industry-standard temptation of using XML as a one-size-fits-all format. Instead, the status file has the form of a simple structured text file, inspired by the Lisp S-expressions, but strictly divided to lines and typeless. This is very easy to generate and parse, especially in shell scripts which form the glue joining the evaluator modules. Moreover, it has the same expressive power as XML, so the files can be converted back and forth if an application wishes.

A typical example looks as in Fig. 2.

As the status files are a relatively recent addition to the Moe infrastructure, they are not yet fully supported by all modules, but they probably will be at the time of publication of this article.

```
task:pyramid                The name of the task [2pt]
lang:c                      Language of the solution
test(                       A section for a single test
    id:1                    Name of the test
    time:0.375              Run time in seconds
    mem:1355776             Memory consumption in bytes
    points:0                Points awarded
    status:RE               Status code
    message:Runtime error   Explanatory message
    exitcode:1              Program exit code
)
test(                       A section for another test
    id:2
    ...
)
...
```

Fig. 2. An example status file.

2.4. *Configuration*

We pay much attention to the configurability of the whole grader. Most aspects of the evaluation process are controlled by many configuration variables, whose values are gathered from several sources and these are stacked one onto another in a manner similar to the Cascading Style Sheets.

First of all, there is a top-level configuration file with global defaults. These can be overridden by per-task configuration files. The per-task configuration usually involves things like setting of time and memory limits, but it can modify any variable if needed. This way, we can extend the compiler options if the task requires a special library to be linked, or change the sandbox options to permit otherwise disallowed system calls. Finally, the settings can be modified for individual test cases.

A fragment of configuration can be also restricted to a specific programming language. This allows compilation commands, settings of the sandbox, or rules for interpretation of runtime errors to be defined differently for each language.

Moreover, the values of the settings are expanded before each use, which includes interpolation of references to other configuration variables. For example, this feature is commonly used to make the compilation command for each language refer to a variable with user-defined compiler switches, or to substitute time and memory limits to the list of sandbox options.

The configuration mechanism also serves as a core of our format of task packages. Essentially, the role of a task package can be played by an arbitrary directory, as long as it contains an appropriately named configuration file. Its variables then point to other files within the same directory, which contain the test cases, judges, model solutions, and other components of the task. As this file naming convention is usually fixed within a single competition, it is customary to use the configuration stacking to inherit most variables from a top-level configuration file and let each task care of its differences from the defaults only (see Fig. 3 for an example).

We are following the discussion on standardization of task packages initiated by Verhoeff (2008), but the Peach format proposed there is too restrictive for our use. We want to

```
IO_TYPE=file                          this is a standard batch task with file I/O
TESTS="1 2 3 4 5"                     names of test cases (files n.in, n.out)
POINTS_PER_TEST=1                     points awarded per test case
TIME_LIMIT=5                          time limit per test case in seconds
MEM_LIMIT=4096                        memory limit per test case in KB
TEST_4_TIME_LIMIT=10                  override for a specific test case
EXT_pas_MEM_LIMIT=8192                Pascal solutions get twice as much memory
OUTPUT_CHECK='$PDIR/judge             this task does not have unique output, so use
$TDIR/$TEST.in                        a problem-specific judge
$TDIR/$TEST.out
$TDIR/$TEST.ok'
```

Fig. 3. An example task configuration file.

keep the assumptions about task formats in Moe at minimum. It is quite possible, though, that a single format will be recommended as a default in the future, when some consensus is reached. Also, Moe's flexibility makes it quite easy to import tasks from other systems.

3. Applications

Our system is still under construction and many parts need lots of improvements. It is however lacking mostly in features and documentation, rarely in reliability. Its design and implementation have already proven itself at multiple occasions.

First of all, it serves as a basis of the competition environment at the Czech national olympiad in last six years. We also regularly use the same environment at the Czech–Polish–Slovak preparation camps whenever they are held in Czech republic. As the organizers of these camps enjoy experimentation with new types of tasks, we have used Moe modules in many previously unexpected ways. A nice example was awarding points by playing a tournament between all pairs of submitted solutions. (More such "hacks" are described in our previous paper.)

A new version of Moe, which pioneered the submitter, has been developed for the Central-European Olympiad in Informatics 2007.

Since 2007, Moe is used as the evaluation back-end of CodEx, which is an automated system for checking students' programming assignments at the Faculty of Mathematics and Physics of Charles University in Prague. To handle this load, we have created queue manager module. As Moe contributes only a small piece to a big puzzle here, we have introduced the status files to make data interchange easier. The CodEx version was also the first to support C# and Haskell.

Recently, the organizers of IOI 2009 have decided to use Moe's sandbox as a part of their contest environment.

4. Future Plans

We plan to continue the development of Moe in the forthcoming years. First of all, we wish to fill all the gaps, especially in the documentation, and make the use of status files systematic. We also want to rewrite the evaluator module and add the feedback, scoring, and supervisor modules as described in Section 2.1.

Further plans include extension of the submitter module to provide on-line feedback, which will make it directly usable in contests like the ACM ICPC. Also, we would like to support more operating systems, architectures and programming languages.

As most free-software projects, Moe is developed by volunteers. Any bug reports, suggestions for new features, patches to the code or any other contributions are heartily welcome. Also, if you use parts of Moe in your contest, please let us know, we are interested in your experience.

References

Evans, Ch. (2009). Linux syscall interception technologies partial bypass. Security advisory CESA-2009-001. Retrieved 27 February 2009 from:
http://scary.beasts.org/security/CESA-2009-001.html

Forišek, M. (2006). Security of programming contest systems. In *Informatics in Secondary Schools, Evolution and Perspectives*. Vilnius, Lithuania.

Mareš, M. (2007). Perspectives on grading systems. *Olympiads in Informatics*, **1**, 124–130.

Mareš, M. *et al.* (2009). *The Moe web site.*
http://www.ucw.cz/moe/

Verhoeff, T. (2008). Programming task packages: peach exchange format. *Olympiads in Informatics*, **2**, 192–207.

Martin Mareš is as an assistant professor at the Department of Applied Mathematics of Faculty of Mathematics and Physics of the Charles University in Prague, a researcher at the Institute for Theoretical Computer Science of the same faculty, organizer of several Czech programming contests, member of the IOI Scientific Committee and a Linux hacker.