Olympiads in Informatics, 2007, Vol. 1, 124–130
© 2007 *Institute of Mathematics and Informatics, Vilnius*

Perspectives on Grading Systems

Martin MAREŠ

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

Abstract. Programming contests often use automatic grading of the submitted solutions. This article describes one such system we have developed for the Czech national programming olympiad, the experiences gathered over the course of its development and also our perspectives on the future of such systems.

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

1. Introduction

Many programming contests in the world, including the IOI, are based on automatic grading of the submitted solutions. This is accomplished by running them on batches of input data and testing correctness of the output. Time and space limits are usually enforced during the process, which allows to judge by not only the (approximation of) the correctness of the solution, but also its time and space complexity.

Multiple such evaluation systems have been developed, but most of them are used only in a single country and they are usually neither publicly available nor well documented. This seems to be a waste of effort by inventing the same things over and over and also by making mistakes somebody else has already made and understood.

This article is our modest attempt to help mapping the landscape of automatic evaluation of solutions. We will describe the MO contest environment we have designed and developed for several Czech programming contests. We will try to review the experiences gathered over the course of its development and present our perspectives on the future of similar systems.

The current version of the MO system can be downloaded from the web site listed in references. At the time of this writing, it was in use for several years in the Czech national olympiad in programming, at the Czech-Polish-Slovak preparation camps (which serve as a training for both our contestants and the system) and also for testing of student homeworks at our faculty. A new version is currently being finished for the Central-European olympiad in informatics (CEOI 2007) .

2. The Contest System

2.1. *Building Blocks*

The MO contest system consists of the following parts:

- *development environment* – editors, compilers, debuggers and similar tools used by the contestants for writing the solutions. In our case, this is just an appropriately configured Debian Linux system with a couple of packages added;
- *submitter* – this is the user interface of the contest system for contestants. It is primarily used for submitting a finished task solution for grading. Currently, we use a simple command-line utility for the national olympiad and a web-based interface for our classes;
- *evaluator* (also known as a *grader*) – takes care of testing the solutions and imposing limits. This is the core of the whole system;
- *feedback* – presents the results of the evaluator to the users. In IOI-type contests, its only role is to generate the evaluation reports and rank lists, but in general it can provide on-line feedback to contestants and/or spectators;
- *auxiliary services* – printing and similar. They vary from contest to contest and there are out of scope of this paper.

In this article, we will focus on the evaluator part, but we will keep in mind its connections to the other parts.

2.2. *Design Goals*

When we were designing our contest system, we had several basic goals in mind:

First of all, the system should be *flexible.* There are many types of contests ranging from the strictly off-line nature of the IOI to those with fully on-line feedback as the ACM ICPC. The variety of contest tasks and their types of interaction is probably even higher. Therefore we should try to expect the unexpected and make the system highly configurable and modular, so that the usual tasks can be dealt with by setting the parameters, while for the exotic ones we can plug in new modules.

Second, we should make the system *secure.* The submitted solutions can try to attack the evaluation system in various ways (not necessarily intentionally). A thorough study of known attacks has been recently published by M. Forišek. Hence the evaluator must be robust enough and isolate the examined solution from the rest of the evaluation system. This is an instance of the classical problem of running untrusted code, frequently studied as a part of OS security.

Last, but not least, the evaluator should be *as simple as possible* (but of course not simpler) in order to allow easy review of the whole code for security and correctness. Because of this, we have avoided putting any user interface into the evaluator and we prefer trivial input and output interfaces instead, which can be then presented to the user by some other components of the contest system.

We have also decided that having the system work on Linux is enough for our purposes. However, most parts of the evaluator can run on any POSIX-compliant system, the only exception being the sandbox, which makes heavy use of special features of the Linux kernel.

3. The Evaluator

3.1. *Sandbox*

The *sandbox* is the core of the evaluator. Its purpose is to run a program in a controlled environment, where both the interaction with other programs and the consumption of system resources (time, memory and disk space) is limited. We use the sandbox for running the solutions being evaluated, but also (with a relaxed set of restrictions) for compiling them.

The implementation of the sandbox makes use of the ptrace interface of the Linux kernel, originally designed for attaching debuggers to programs. It runs the program within a separate process and asks the kernel to interrupt the process every time it tries to make a system call. The sandbox then examines the parameters of the call and either lets the program continue its execution, or terminates it. For example, allocation of memory is always allowed, opening of files is allowed after inspecting the path to the file, and creating a new process is always forbidden.

This approach has been well studied by the secure system researchers and it has several known drawbacks, mostly related to race conditions in multi-threaded programs (another thread can modify the parameters of a syscall in the small time window between checking the parameters by the monitor and really executing the call). Fortunately, none of these problems apply to our case as no concurrency takes place. Another thing needing some extra attention is that although the contestants are not expected to use any syscalls outside the basic reading and writing of files and allocating memory, the standard libraries they call do use much more. We can however still manage with a simple list of obviously safe syscalls and a handful of easily verifiable exceptions.

Consumption of resources is controlled in the usual ways. We use the ulimit mechanism provided by the kernel whereever it is possible, that is to constrain the memory (address space) allocated by the process and also the maximum number of file handles used. Disk space filled by the program is limited by a disk quota for a special user ID which is used exclusively for the sandbox.

Limitation of execution time is slightly more complex, because it is not obvious what exactly should the time mean. The sandbox allows to measure either the user time of the process (which is accounted by the kernel and includes only timer ticks spent in the specific process in the user mode of the processor, that is, excluding syscalls and interrupts) or the wall clock time (as reported by the real-time clock of the OS). In both cases, the sandbox monitors the state of the timer periodically, kills the process when it exceeds the allowed time and it also checks the exact value of the timer when the program finishes successfully.

We currently use the first method for almost all tasks, since it makes the timing of the program less dependent on other programs running on the same system (but not completely independent, see the discussion below). As the kernel measures the user time by sampling on timer ticks, this method can be circumvented by processes which tend to work in short time quanta and sleep in the meantime, but in our case no syscalls for sleeping should be available.

The second method is sometimes used for interactive tasks, where substantial amounts of time can be spent by waiting for the system's reply and a deadlock is possible when violating the protocol. Proper accounting for communication delays and deadlock detection are obviously preferable, but they are not always easy to perform.

3.2. *The Scripts*

Except for the sandbox, the rest of the evaluator is implemented as a set of scripts for the Bourne shell. This can sound strange at first, but as most of the job is just gluing small parts together, the shell is often a better tool than a "real" programming language.

As we have already mentioned, the design is modular. We have a library of shell functions serving as building blocks for performing the basic tasks: compilation of the contestant's solution, preparing inputs, running it inside a sandbox, fetching and validation of the program's outputs. The evaluator itself (the front-end used by the organizers to process the solutions) is then a simple script which just calls the building blocks in the right order.

All modules also record whatever they are doing in a common log file, which can be later examined by the contestants to understand what they have done wrong.

3.3. *Configuration*

All the building blocks are highly configurable. The configuration variables range from simple settings like the table of compilers and their options for various languages, or time and memory limits, to commands run to perform various tasks (for example verification of correctness of program output).

The configuration files are again shell scripts. Their primary task is to set environment variables corresponding to the configuration settings. First, a global configuration file for the contest is loaded (it usually defines the basic parameters like the compilers and also provides defaults for all other settings), then it can be modified by per-task configuration and finally even individual test cases can override anything (for example, different time limits can be used for different test cases).

3.4. *Task Types*

The standard building blocks know how to handle the usual types of contest tasks. If the task is of one of these types, it is sufficient to set the corresponding parameters in the configuration. You can of course define your own task types by providing a couple of shell functions or scripts.

The most common task type is the *input/output task.* The tested program is given an input file and it produces an output file. Alternatively, communication by standard input and output can be used. A judge program specified in the configuration is then run to check the correctness of the output and its exit code determines the outcome. The default judge is just a call of the diff utility, set up to compare the file with the (unique) correct output, ignoring differences in white space. The judge can also write messages to its standard error output, which become a part of the evaluation log.

Other tasks can be *interactive.* Such tasks are for example games, which communicate on-line with an opponent played by the judge. In this mode, the evaluator runs the tested program and the judge in parallel and it connects the standard output of the program to the standard input of the judge and vice versa. The exit code and the error output of the judge are again used to determine the outcome and log messages. We usually wrap the protocol between the program and the judge in a library linked to the evaluated programs, so that the contestants do not have to take care about details of the communication protocol and proper flushing of I/O buffers.

The third group of standard tasks are the *open-data tasks,* in which the testing data are public and the contestants submit only the output files, which they can obtain in whatever way they wish. In this mode, the modules taking care of compilation and running of the solution are replaced by a simple fetching of the submitted output file and the file is then processed like in an I/O task.

3.5. *Hacks*

The flexibility of a system can be probably best judged by its applications to situations unknown at the time of its design. Here we show a couple of such "hacks".

At one of the previous preparation camps, we had an *approximation task* and the points were determined by the quality of the approximation (as we did not know the exact optimum, the quality was measured relative to the maximum of the best of the contestants' solutions and our program). This does not fit well the structure of the evaluator, because it is run separately for different contestants. However, we can take advantage of the simple interface between the evaluator and the rest of the contest systems – for every contestant, the evaluator creates a simple text file containing points and judge's verdicts for all test cases. Hence we can let the judge check the validity of the output and record its value in the verdict and after evaluating all contestants plug in a simple program, which will read all verdicts and recalculate the scores appropriately.

This method can be also used for *grouping of test cases* (we want to avoid awarding points to programs which always print "No solution", so we combine test cases of roughly the same complexity to groups and award points only if the whole group is answered correctly). We let the evaluator assign points for individual test cases and then a grouping module is run, which calculates the group scores. As the grouping technique is becoming commonplace, this module will be moved to the library of standard modules soon.

Another interesting application is *pitting* the solutions against each other if the task is a two-player game. Instead of playing against a judge provided by the organizers, we simulate a tournament in which all possible pairs of matches are played and then the points are assigned according to the outcome of the tournament. This again does not fit in the framework directly, but we can replace the default interface of the evaluator by a simple program (in fact, it was something like 30 lines of shell script), which will use the standard modules to compile the programs, run them in the respective sandboxes and connect them together through a judge, which will make sure that everybody follows the rules of the game and the communication protocol.

4. Perspectives

Our contest environment has proven itself useful in multiple contests, but it is far from being the final word on the subject. Several questions keep arising and the answers will shape the future contest environments (and also future versions of ours).

4.1. *Time Measurement*

The computers are becoming gradually faster and the traditional one-second resolution of time limits requires still larger input data to distinguish between efficient and slow solutions. Also, the speed of the processor is increasing faster than the speed of disks, so with larger inputs the proportion of time spent by reading the input increases.

The obvious solution is of course increasing the timer resolution and use sub-second timing, but it is not so easy as it might seem, because there is a lot of noise in the time measurements, which suddenly becomes very visible on this scale. Many different factors contribute to the noise, the most important of them being caches (both the code and data caches of the processor and the disk cache of the operating system). The initial contents of caches when the program is started are unpredictable and the algorithms controlling cache operations are usually very complex and they involve hidden variables. For example, most data caches are set-associative and indexed by physical addresses, so the efficiency of the caches is influenced by placement of the pages of memory allocated by the program in the physical address space of the processor. While it is very rare for the cache effects to cause slowdown on the order of magnitude (even this has been observed, but not in a contest task), the effects are large enough to become a significant factor in millisecond time measurements.

We can try to use the standard engineering techniques to deal with the noise: repeat the measurements several times and take a minimum or an average of the values, or try to make the initial state more predictable (e.g., by letting the evaluator pre-read the input file to increase the probability of it being present in the disk cache; by the way, in our evaluator this is a pleasant side-effect of copying the input file to the directory accessible to the sandbox just before running the program). This helps to eliminate the most visible effects, but definitely not all of them, since the physical addresses used in the different testing runs will very likely be correlated.

An interesting approach has been recently suggested by Szymon Acedanski and tested at CPSPC 2006 in Warszawa. Instead of measuring the time, we can count the number of instructions executed in the user mode of the processor. This can be easily accomplished by a simple kernel patch using the performance counters of the processor. The instruction count corresponds to the execution time on an idealized computer and it is not influenced by any cache effects nor other sources of noise, so the resolution can be arbitrarily fine. The only drawback is that it could hide more implementation details than we would wish – for example, this model makes integer addition and floating-point multiplication equally expensive, which might not be desired.

We plan to implement this type of timing in our environment to give it more field testing.

4.2. *Inputs and Outputs*

Large input files are necessary not only because of precision of timing, but also when we want to distinguish between similar time complexities, especially between linear and linear-logarithmic one. As already mentioned, this leads to a big fraction of time being spent on parsing of files and it also significantly hurts contestants using the slower I/O libraries (for example, we have seen several tasks which are impossible to solve if one uses C++ streams, because the stream library is unable to read the input within the time limit).

Experienced authors of tasks often take this problem into account and they use tighter encodings of inputs, like describing a tree by its traversal sequence. This helps in some cases, but it is far from being a universal technique. It can also unnecessarily complicate parsing of inputs.

One possibility (again suggested by Szymon Acedanski) is to ask the contestants to use a special library for reading the input, which provides functions for reading values of all standard types. These functions consume input files preprocessed to a special binary format, which saves most of the reading and parsing overhead. However, the contestants have to learn the new functions.

A similar effect with less complications for the contestants could be achieved by making the evaluation system parse the input before the clock starts and providing it to the program in global variables. The program can then access its input data by simply linking with a library. This trick of course leaves out an interesting class of problems – the streaming problems, where the input is larger than available memory and it has to be processed sequentially. On the other hand, we can view such problems as a special case of interactive tasks and also hide the implementation of input and output in a library.

It is not clear if this format of input is the best answer to the problem, but it is definitely worth trying and we plan to experiment with it in our framework in the near future.

References

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

Mareš, M. (2007). The MO-eval web site.
`http://mj.ucw.cz/mo-eval/`

M. Mareš is a doctoral student 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.