# Automatic grading of programming exercises in a MOOC using the INGInious platform

Guillaume Derval, Anthony Gego, Pierre Reinbold, Benjamin Frantzen and Peter Van Roy, *Université catholique de Louvain, Belgium*

ABSTRACT

**This paper represents a return of experience about the use and design of an external grader for a computer science MOOC. We introduce INGInious, an automatic grader for both on-site students and students from the edX platform. Then, we explore both the practical and technical approach of its use. We see that providing good feedback is a key element to learning, and that it must be as detailed as possible, but also, because of theoretical limits, transparent. Then we see how a good external grader must be designed, that is, generic, safe and reliable.**

## Introduction

When learning computer science, it is important to give actual programming exercises, where students are required to write fully working algorithms. It has been long since the idea of automating corrections of such exercises has been suggested, but faced different limits, theoretical or technical, resulting in full or partial involvement of a human in the correction.

In the case of MOOCs, the automation of these corrections is no longer a desire, but also a requirement. The number of students is usually too great to permit handmade corrections. Even with automation, the slightest involvement required from a physical person can become huge and induce great costs. Apart from alternative forms of feedback [Nicol, 2010], the process therefore needs to be fully automated. This paper presents methods and tools that allow automatic grading of code made by students.

INGInious [Derval et al., 2014] developed at the Université Catholique de Louvain by Guillaume Derval, Anthony Gégo and Pierre Reinbold, is a tool designed for this goal. INGInious is made to be generic, capable of running virtually any programming language; safe, running students' algorithms in specific virtual environments confined from the base system; and scalable, capable of dealing with large numbers of submissions. In addition to grading algorithms autonomously, INGInious also provides an interface with the edX platform. INGInious is a free and open-source software (FOSS) distributed under the AGPL license. Its sources are available on GitHub.

In this paper, we first discuss the use of INGInious in the Programming Paradigms course given on edX by Peter Van Roy (Louv1.1x [Van Roy, 2014a] and Louv1.2x [Van Roy, 2014b]), and then we present key elements of its architecture, its safety, and its scalability.

## Creating exercises with INGInious

To prepare INGInious for correcting student exercises, two preliminary steps are needed. First, it must be installed, which requires some configuration. More information about this important step is available in the documentation of INGInious.

Second, courses and exercises can be added. INGInious classifies exercises in terms of courses and tasks. A course contains multiple tasks, and a task contains one or more exercises. Each exercise has the form of a directory, containing at least two files. One of them contains various information about the exercise, like its name, a description, and various constraints to be applied when tested (time or memory limits). The second file is the actual script to be run. This script can in turn call various components or subscripts present in the virtual machine (we discuss these virtual machines in the section Secure code execution and environments).

This script is intended to check the student's answer and provide some feedback. The feedback must provide at least two elements, as edX course designers will know, that is, a binary value to state if the exercise is passed or not, and an arbitrary text to provide the user some feedback. This feedback is discussed in the next section. At this point, INGInious is ready to accept student input. When adding an exercise to an edX course, a (small)

additional step is required, that is, setting up the exercise in edX (see edX documentation about external graders [edX, 2014]).

# Providing the user with good feedback

## Introduction

In the context of MOOCs, one must keep in mind that, historically, automated feedback was the only one the students were going to get, at least in large-scale courses. Nowadays, other forms of reviewing have emerged (peer reviewing for instance), but automated feedback can still be wished for, especially in hard science, due to the factual nature of most exercises. If we choose to use it, it is important to help students to find, understand and correct their mistakes themselves, in order to maximize their learning [Nicol, 2010]. A good feedback is therefore a requirement of a good grading process. Giving a result as 'passed' or 'failed' is insufficient.

## Cause of errors

When writing programs, there are three different kinds of errors.

(1) The program is not correct with respect to the programming language, that is, there is a compilation error.

(2) The program is syntactically correct but the execution encounters an unexpected operation, that is, there is a runtime error.

(3) Finally, it is correct regarding to the programming language, but does not provide the correctanswers.

One should note that each of these errors does not necessarily mean that the student has a comprehension issue. In our case, a student could very well understand a concept, but fail to apply it to a concrete example because of the particular syntax of the programming language used (in our case, Oz). However, it is quite useless in computer science to understand a concept and not being able to use it, so helping the student to correct the mistake is important.

## How feedback is provided

Computer scientists know that automatic correction is not an easy task, because of theoretical limits: it is actually impossible to check whether the student's code does the same as a correct one (consequence of Rice's Theorem, see [Beckman, 1980]). There are different approaches to check for algorithm correctness. Some mathematical theories provide actual tools to prove a program's correctness (for example, see [Vander Meulen, 2014]). However, these can be quite complicated, and represent much trouble to implement. Moreover, in the context of a Programming Paradigms course, the goal is not for students to ensure program correctness, which is a theoretical field in itself, but rather to understand the concepts exposed. We therefore take the approach of unitary testing. In other words, we run the student's code with some inputs and check the output. For each test, we then handle the errors.

## Classification of feedback

We can reduce all three kinds of errors to one behavior in the grading process. First of all, before even thinking of interpreting the error, we must accept that it is probably not possible to predict all the possible answers a student could give, and the reasons he gave them. In this regard, it is good practice to be as transparent as possible in the errors detected. It is important to tell the student how his algorithm has been tested, that is, with what input, so that he can test it himself, and reflect about the error.

Nevertheless, we would like to provide more precise feedback, at least for the most common errors. Concerning compilation and runtime error messages, they are often generic and sometimes quite meaningless, even though they often happen for the same reasons. It is usually beyond the scope of a compiler to provide advice to correct errors, but a grader can suggest different leads to the students when a particular error is encountered. Also, it is possible to do some static checks to detect most common syntactic and conceptual errors. These errors are often similar in all possible exercises, and will most likely not be reproduced once understood. Other errors can be specific to a particular exercise. Here, static checks can also be performed. But with a bit of practice, it is possible to recognize frequent mistakes made by students, resulting in the same wrong output, or an error. When this error is detected, it is a good idea then to give more information to the student and probable cause of mistakes.

## Practical experience

There are some additional thoughts one should keep mind when designing exercises. First, it is very easy to see the limits of unitary testing: if an algorithm is supposed to compute, say, '4+2', one can design an algorithm that will compute '1+5', '3+3' or simply return 6 directly. Each of these algorithms provides the good answer, but the

computation is not correct. Combined with the transparency principle, it is easier for a student to cheat.

This is an issue for on-site students of a MOOC. The idea is thus to make it more difficult for the student to cheat rather than giving the good answer. There are different possibilities to achieve that. A good start is to make many different tests. Other ways can be to test subsets of the algorithm (for instance, each function individually), force students to use some pre-defined functions, or pre-defined lines of code, and finally, perform some static check to prevent some specific calls or patterns.

More optimistically, though, a student could also write an algorithm that works in some cases but not in others, for instance, he might not even know why his algorithm works or doesn't work. Our goal here is to provide feedback that will help him understand. Among these test cases, we will therefore test limit cases of the algorithm, that is, cases that can be somewhat singular and require additional steps from the algorithm (for example, trying to sort an empty list). Students often forget about these limit cases, and they often are the reason the algorithm is incorrect: a forgotten limit case can often cause the algorithm to crash. Also can be tested some cases that are to be considered typical regarding the goal of the exercise, to make sure the student really understands its purpose. Once all these tests pass, we can consider that goal achieved, and it will often be enough to consider the exercise correct.

Finally, to avoid too many different causes of mistakes, which complicate the design of the grading process, and most likely the understanding of students, it is a good idea to keep exercises small, and focused on one particular aspect. The automation of corrections being as it is, correcting big algorithms gives more opportunity to encounter an unexpected situation, and thus provide wrong feedback, which often irritates students. Small exercises will allow them to test their knowledge in a straightforward manner, and identify their errors quickly. More elaborate exercises can be done at different intervals, and are an opportunity to cross their knowledge and challenge them more, but it is a better practice to do that only when we are sure each concept has been understood individually.

## Architecture of INGInious

INGInious is composed of two parts, called the *frontend* and the *backend*. The *frontend* provides a simple web interface for the students that do not follow the MOOC (at UCL, INGInious is used for other courses that are not on edX), an administration that allows writing tasks directly in the web browser. The frontend also manages the database (using MongoDB) containing submissions and statistics.

The plugin system is also managed by *the frontend*. It can be configured to run the edX plug-in, communicating with the edX platform and allowing to grade code using the simplicity and power of INGInious on edX. The edX plugin is in fact, from the point of view of the edX platform, a passive grader. The configuration of this plugin is very simple and is described in the documentation of INGInious. Other plugins are available, allowing calling the grader externally via JavaScript for example, or providing additional statistics on the submissions.

Made as an independent library, easily reusable, the backend creates, manages and deletes containers, running the code made by the students. The backend does not store anything in a database or on the disk. Figure 1 shows a simplified diagram of the INGInious architecture.

Once the environments (container images, described in the next section) and the tasks' scripts have been written by the teacher or its assistants, everything is automatized: the students' code will

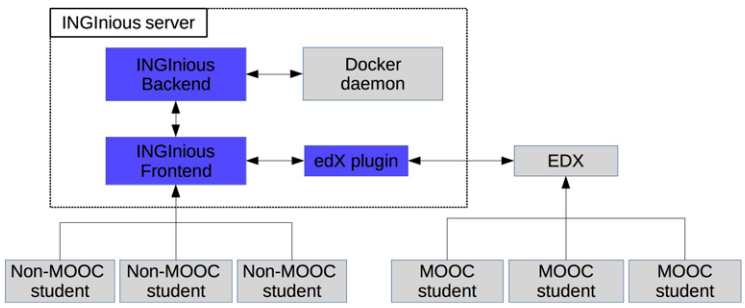


Figure 1. Simplified architecture of INGInious

automatically be graded on submission, without additional steps from the teaching staff.

Students have two ways to interact with INGInious: the first being the frontend web interface, that needs the creation of an account (at UCL, accounts are provided via an LDAP server); the second is the edX plug-in, which receives anonymous requests from the edX platform. The use of this plug-in is completely invisible for edX students. The course can be followed simultaneously by on-site students, each with its own INGInious account; and by edX students, each with its edX account, whom will post their answer directly at edX.

## Secure code execution and environments

INGInious must execute its code in a safe environment that allows both resource control and safety. Code submitted by students should not be trusted in any way. The separation between the host machine and the safe environment is traditionally handled by the usage of virtual machines (Pythia [Le Clément de Saint-Marcq & Combéfis, 2012], the predecessor of INGInious, used User Mode Linux, a kind of virtual machine), that in fact separate the untrusted processes via a hypervisor. This hypervisor manages the resources available to the virtual machines.

This approach certainly has advantages (nearly complete separation of the processes), but also has some drawbacks, in particular creating new environments is hard and there is a huge overhead due to the presence of a second operating system.

INGInious uses another approach, based on Docker [Docker, Inc., n.d.-b], a relatively recent technology that provides containers. Containers are safe and closed environments, but run inside the host kernel. This resource separation is in fact provided by two features of Linux, kernel namespaces and cgroups. Containers are therefore very lightweight: they do not start and manage another operating system, as, for the host OS, containers are simply constrained processes. Like virtual machines, containers can be run with a specified amount of memory, be associated with specific CPUs, and mount shared folder from the host.

Moreover, Docker provides a simple way, called Dockerfiles [Docker, Inc., n.d.-b], to create new container images (or environments). With a very simple syntax, anyone that uses a Linux command line tool can quickly create a new environment. Code listing 1 below is the complete content of the Dockerfile used to generate the container for the Programming Paradigms course. It is somewhat simple: it inherits from the default INGInious container image, and installs Mozart 2 (which provides the Oz language, used for the course).

```
# Inherits from the default container FROM
ingi/INGInious-c-default
# Add the Mozart 2 rpm to the container
ADD mozart2-2.0.0-alpha.0+build.4105.5c-
06ced-x86_64-linux.rpm /mozart.rpm
# Install dependencies of Mozart 2 and Mozart 2 RUN
yum -y install emacs tcl tk
RUN rpm -ivh /mozart.rpm RUN rm /mozart.rpm
```

Code listing 1. Dockerfile for ingi/inginious-c-oz

Docker uses a union file system to store the container images. The container images are viewed as layers, and each command in a Dockerfile (from inheritance to RUN commands) creates a new layer. This allows to drastically reduce space used to store the different environments, as only the difference between the custom container images and the default one is stored.

INGInious launches a container each time a submission is made, from the appropriate container image. Containers are then very short-lived (they live the duration of the grading). Container images are most of the time shared between multiple tasks or even between multiple courses. For the purpose of the MOOC, only two container images were used, *ingi/inginious-c-oz* (that provides Mozart 2), and ingi/inginious-c-pythia1compat (that provides a compatibility layer for Pythia). Other container images are available, allowing using different languages: C++, Java 7/8, Scala, Python 2/3 are also provided with the sources of INGInious. Creating new containers to add new languages is very simple as shown earlier.

This way to use containers means that there are as many running containers as running tasks. With the union file system, the disk is not used by more

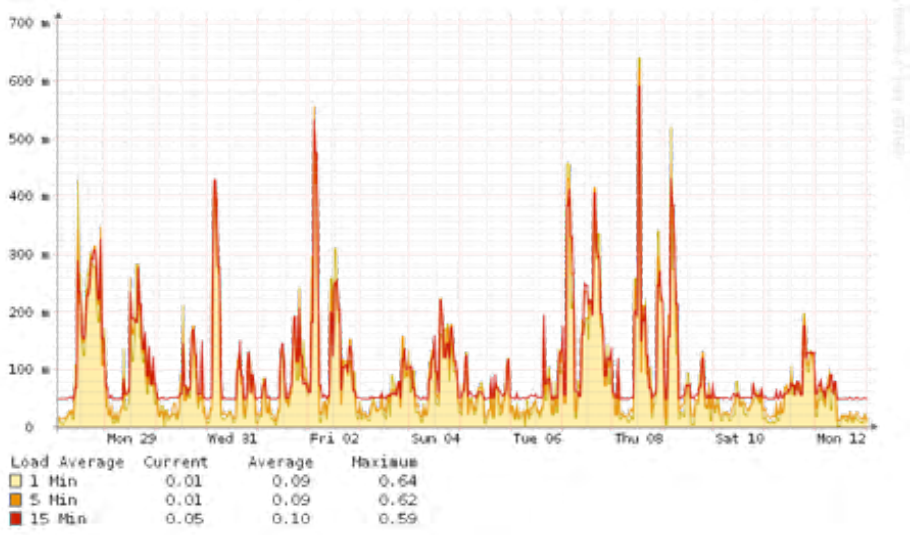


Figure 2. Load averages during the final exams of the MOOC

than a few kilobytes at each container start. There is also a very low memory footprint. The kernel scheduler then handles the repartition of the CPU resources between the different running processes. INGInious adds some constraints, such as timeouts, to the container to ensure that they correctly end. Processes that exceed the time allowed to them or that use too much memory are killed, sending an appropriate message to the student.

## Scaling of the machine

At INGI, we use a single machine running all the components of INGInious. We have currently four courses that use INGInious: *the Programming Paradigms MOOC, Computer Science I, Artificial Intelligence and Advanced algorithms for optimization [Deville, 2014].*

Programming Paradigms has a broad audience compared to the other courses: in the Fall 2014 edition more than four thousand students enrolled in the second part of the course. The other courses count between thirty and four hundred students. Exercises of the courses Programming Paradigms and Computer Science I asks students to write very small algorithms that are never long to complete, and do not involve a lot of computing. The two other courses run exercises that may take more than five minutes of computation to complete.

The machine hosting INGInious was scaled to handle these two types of loads: a large audience running lightweight exercises and a small audience running computation-intensive exercises. We used a virtual machine with 6 (virtual) CPUs and 12 GB of memory. The figure 2 represents the load averages between December 28th, 2014 and January 12th, 2015, which are the days when the final exam of the MOOC occurred. All the other courses were already finished. During this period, around 600 students made 5976 requests to INGInious, with an average running time of 5.17 seconds. The maximum number of concurrent requests was three.

From the point of view of the MOOC, the machine is clearly over-scaled, with the load averages often below one (less than six times the maximum sustainable load), and with memory usage always under 30%. A small dual-core virtual machine with 6 GB of memory is sufficient to handle the MOOC load. INGInious thus does not need to run on a costly cloud.

The machine we use is well scaled for the usage of INGInious made at UCL, as the two more computationally intensive courses sometimes put it at its limit. Usage peaked twice during the Fall 2014 semester, at the deadlines of the course advanced algorithms for optimization, with a load average of 6 for some hours. The management of the peaks is still a work in progress in INGInious, as heuristics for queue management have to be developed.

## Conclusion

Automation of correction is still a great challenge nowadays, and the stakes are high in the context of MOOCs. In the future, with the increase of use of ICTs, more and more students are likely to follow courses online, both in a formal, legal, or informal manner. Followed by actual students seeking for a certificate with a legal value, automatic correction is a key element of checking their competences. But more than just a corrector, an automatic grader has

also to be seen as a feedback provider, which can analyze student's understanding and guide them in their learning.

Different tools make it easier today to design such grader, with different functional requirements in mind (genericity, safety, reliability). INGInious is one of these graders, and provides both easy-to-use features (for students and instructors) and these requirements. The use of INGInious is very easy, whether for automation of correction, but also by executing about any task based on a student input. More than its use in the context of MOOCs, many prospects therefore rise, that might change the vision of online learning itself.

The use of such tools will indeed change the way instructors are able to design their courses, and provide a very personalized learning despite the distance and virtualization, and the prospects in this regard are most likely to be as great and exciting as the future.

## References


- **Bonaventure, O., Combéfis, S., & Pecheur, C.** (2014). LFSAB1401 – Informatique 1 [course].
- **Beckman F. S.** (1980). Mathematical Foundations of Programming, The systems programming series, Addison Wesley.
- **Derval, G., Gégo, A., & Reinbold, P.** (2014). INGInious [software]. Retrieved from https://github.com/UCL-INGI/INGInious
- **Deville, Y.** (2014). LINGI2261 – Artificial Intelligence: representation and reasoning [course].
- **Docker, Inc.** (n.d.-a). Dockerfile reference. Retrieved from https://docs.docker.com/reference/builder/
- **Docker, Inc.** (n.d.-b). What is Docker ?. Retrieved from https://www.docker.com/whatisdocker/
- **edX.** (2014). External Grader. Retrieved from http://edx-partner-course- staff.readthedocs.org/en/latest/exercises_tools/external_graders.html
- **Le Clément de Saint-Marcq, V., & Combéfis, S. (2013).** Pythia [software]. Retrieved from http://www.pythia-project.org/
- **Nicol, D.** (2010), From monologue to dialogue: improving written feedback processes in mass higher education, Assessment & Evaluation in Higher Education, Vol. 35, No. 5.
- **Vander Meulen J.** (2014). LINGI1122 - Program conception methods [course].
- **Van Roy, P.** (2014a). Louv1.1x Paradigms of Computer Programming – Fundamentals [course]. Retrieved from https://www.edx.org/course/paradigms-computer-programming-louvainx- louv1-1x
- **Van Roy, P.** (2014b). Louv1.2x Paradigms of Computer Programming – Abstraction and Concurrency [course]. Retrieved from https://www.edx.org/course/paradigms-computer-programming- louvainx-louv1-2x
- **Schaus, P.** (2014). LINGI2266 – Advanced Algorithms for Optimization [course].