An Infrastructure for Mechanised Grading

Christian Queinnec

AN INFRASTRUCTURE FOR MECHANISED GRADING

Queinnec Christian
LIP6, Université Pierre et Marie Curie, France
Christian.Queinnec@upmc.fr

Keywords: Mechanised grading, grading experimentation

Abstract: Mechanised grading is now mature. In this paper, we propose elements for the next step: building a commodity infrastructure for grading.
We propose — an architecture for a grading component able to be embedded into various learning environments, — a set of extensible Internet-based protocols to interact with this component, and — a self-contained file format to thoroughly define an exercise in order to ease deployment. The infrastructure was designed to be scalable, robust and secure.
First experiments with a partial implementation of this platform show the versatility and the neutrality of the grading component. It does not impose any IDE and it respects the tenets of the embedding learning environment or course-based system.

History

Around 2000, with the help of some colleagues from UPMC, we set up two experimentations involving mechanised grading. In the first experiment (reported in (Brygoo et al., 2002)), we adjoined to the DrScheme programming environment (an Integrated Development Environment (or IDE) devoted to the Scheme programming language (Findler et al., 2002)) a mechanised grading facility: the student chooses an exercise, reads the stem, types then debugs his program and finally hits the "Check" button to obtain a mark and a page explaining the tests that were performed and led to that mark. This mechanised grader is a stand-alone plug-in to DrScheme; every year since 2001, around 700 students have had used it on their home computer (night and day) or during lab sessions.

In the second experiment (reported in (Queinnec and Chailloux, 2002)), we organised general programming examinations or contests where we had several hundreds of undergraduate students to grade in a short time. Centralised mechanised grading was our sole option. Therefore we wrote unit tests to check students' programs considered as black boxes. Soon afterthat, we designed a framework and implemented some libraries to ease the production of mechanised graders that is, specialised programs that grade students' files for a given exercise. From 2001, we have been grading several hundred examinations per year.

Around 2005, our various experiences with different programming languages such as Scheme, C, Ada, PHP, Perl, Java, make or shell convince us that mechanised grading was mature enough for courses that mainly rely on some programming language(s) both for exercises and/or examinations. We proposed to unify our previous experiments into a generalised framework and a multi-language grading architecture to ease the production of graders and to lessen the associated chores. Our goal was to build

1. an autonomous and scalable grader that may be embedded within various systems,
2. REST-based (Fielding and Taylor, 2002) protocols to interact with this grader,
3. a file format for self-contained deployable exercises.

While building this grader, we added new goals in order to provide a grading infrastructure that is, grad-

ing as a software service (SaaS).

We will present, in the first section, the advantages of mechanised grading and some of our goals. The second section will describe an overview of the architecture, nicknamed FW4EX (for“ framework for exercises”), and its prominent features. In the third section, we will summarise the results of our lattest experiments using the FW4EX architecture. Related works is addressed in the fourth section.

1 Mechanised grading

Writing a program is a complex activity requiring

1. to understand a specification,
2. to imagine a solution,
3. to write it down in some programming language and within some environment development (IDE),
4. finally to test it to make sure it complies with the specification.

Any problem that arises during this life-cycle forces the student to re-iterate parts of this cycle. A way to assess this skill is to let students program with professional tools and to check programmatically whether their programs comply with the specification. Professionally, this is called “unit testing” and popular frameworks such as JUnit[1] were developed for that activity. To use professional tools and to obey professional rules was something we wanted our students to be exposed to.

There are some differences though with usual unit testing.

- Unit testing is designed to give a binary answer: 1 (pass) or 0 (fail). In a grading context, we want to give a more representative and accurate mark for the student's work say a mark between 0 and 20 (as usual in French universities). This may be done by multiplying the tests and combining their partial results.
- Unit testing frameworks often depend on one programming language. However some specifications leave the choice of the programming language to use up to the student. Therefore, a grading framework has to cope with multiple languages.

To design mechanised graders is a hot topic as illustrated by the numerous papers presented these last years like Quiver (Ellsworth et al., 2004), RoboProf (Daly and Waldron, 2004), TorqueMOODa (Hill et al., 2005), gradem (Noonan, 2006), APOGEE (Fu et al., 2008). An overview of this topic may be found in (Douce et al., 2005a). Among well known systems containing graders are CourseMaker (Higgins et al., 2005), BOSS (Joy et al., 2005) and now ASAP (Douce et al., 2005b).

Mechanised graders offer multiple advantages:

- consistency: students are graded uniformly (and anonymously) without the biases due to (multiple) human grader(s).
- fairness: if one student finds a problem in the specification or the associated tests then all students benefit from the upgraded grader.
- persistence: once created, assessments are forever available for students who may practise them whenever they want, in the same conditions the assessment was initially offered. We dubbed this “dynamic annals” (Queinnec and Chailloux, 2002).

Many colleagues have a negative feeling against mechanised grading. Unit tests consider programs as opaque black boxes; quite often (but see (Higgins et al., 2005), (Joy et al., 2005)) they do not consider or appreciate style, performance, good algorithmic usage, etc. We think that some aspects of this critic are due to youth problems:

- Style checkers (CheckStyle[2], PMD[3], FindBugs[4] for Java, Perl::Critic[5] for Perl, etc.) exist and are more and more professionally used. To incorporate them in unit tests will lessen this critic. Performance may also be checked given appropriate environment setup and non-toy problem size. However these improvements will dramatically increase grading duration.
- Since we collect all the answers to our assessments, we expect to be able to build a taxonomy of common errors and to mechanically recognise instances of these errors in order to provide appropriate answers.
- Unit testing may produce very detailed reports of all tests whether they passed or failed. To analyse these traces is a useful skill to be acquired by students (as elsewhere observed, good debuggers are likely to be good programmers while the inverse is less than true) if some familiarity with these tools is organised ahead of assessments.

Therefore a grading component must be able to grade exercises in various programming languages,

[1] www.junit.org

[2] checkstyle.sourceforge.net

[3] pmd.sourceforge.net

[4] findbugs.sourceforge.net

[5] perlcritic.com

various settings: complete programs (stand-alone, client or/and server), program fragments (function, method, class). A grading component must be able to be put to work as part of a learning environment or other systems: it must not dictate whether exercises are summative or formative, it must be as independent as possible from scholar databases.

Of course, it must also be simple to use, robust in face of crashes, secure (respect privacy and anonymity, resistant to pirates and malicious student's works). Finally, our dreamt grading component also ought to be scalable in order take benefit of new technologies such as cloud computing (Amazon EC2 for instance) or Internet-based storage (Amazon S3 for instance) if demand for grading should increase.

2 The FW4EX project

The FW4EX project aims to build such a grading component under three additional constraints listed in this Section. These are:

- **Exercises must be easily deployed.** Copying the artefact defining an exercise into an appropriate directory should be sufficient. This requires a precise format for artefacts defining exercises and their whole life-cycle.
- **IDEs should be FW4EX-clients.** Students should program with appropriate tools. Using an IDE is the current way of programming therefore IDEs must be able to propose exercises, gather student's files, submit them for grading and display grading reports.

  Fortunately, IDEs (Eclipse, NetBeans, Scintilla, Emacs, etc.) often offer a plugin architecture that makes easy to add a button or a menu item for that task. In order to simplify these plugins, interactions with the FW4EX-related servers only use the HTTP protocol and more precisely REST-based style (Fielding and Taylor, 2002).
- **Segregate functionalities into separate components.** FW4EX-related functionalities are componentised so they may be organised into appropriate workflows (including human reviewers for instance). Moreover these components should cope with firewalls, university-imposed authentication method, provide skinning and be scalable. Given that these components are accessed via HTTP, they are called servers and altogether form the building blocks for a grading infrastructure.

  These servers gather related functionalities such as serving exercises (*e* server(s)), acquiring student's files (*a* server(s)) or, serving grading reports (*s* server(s)). A single physical machine or a cloud of virtual machines may implement all or part of these logical servers depending on the topological constraints, the expected load or throughput, the intended availability and the configuration of the workflow.

We think that these three goals provide a strong basis for a universal and versatile grading infrastructure. This infrastructure takes care of storing/retrieving exercises, accepting/grading submissions, storing submissions and their associated grading reports thus relieving teachers from all these chores. FW4EX aims to be a grading infrastructure in the line of SaaS (Software as a Service).

This infrastructure provides many opportunities for the creation of appropriate eco-systems as:

- Authoring tools to build artefacts defining exercises.
- Teaching tools to design sets (scenarii) of exercises, analyze answers, compute statistics, etc.
- Deploying tools to instantiate networks of servers to operate within a university (with a single physical server, serving without any authentication, all aspects of FW4EX) or within a cloud for a worldwide company where scalability, latency and accountability are of paramount importance.

To achieve these goals, we offer an artefact format for exercises (see Section 3.1), a series of HTTP protocols (Section 3.4) and some returns of experience (Section 4).

3 The FW4EX architecture

The FW4EX framework takes benefit of our previous experiments and tries to address a variety of use cases. In this Section, we present the main lines of the architecture, the protocols and the format of exercises.

Figure 1 shows a simplified sketch of the logical architecture of the FW4EX platform. This architecture supposes the presence of Internet and heavily relies on REST-based services (Fielding and Taylor, 2002).

From the student's point of view, after choosing (or be assigned) an exercise, his browser (or any FW4EX-compliant client, see Section 3.4) fetches a zipped file (from an exercises server *e*) containing the stem and all the necessary files or documents required to practise the exercise. When ready, the student submits his work (to an acquisition server *a*) where it will be picked by a grading server *gd* that will instantiate an appropriate grading slave (m_1 or m_2) to grade the

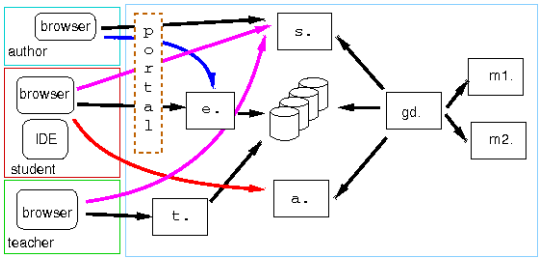

Figure 1: Sketch of the logical architecture of the FW4EX platform. The arrow tells which machine initiates connections.

student's files. Once graded, a final report (an XML file) is made available to the student (on some storage server *s*) as well as its initial submitted work.

Exercises servers *e* only serve exercises i.e., static files. Exercises servers may be implemented with Amazon S3 technologies. In our experience, these servers are often proxyed via a portal (see Figure 1) allowing students to choose exercises among the set prescribed by their teachers. The portal also skins the exercises with the university's web conventions (colors, logos, authentication, etc.).

Acquisition servers *a* are stand-alone servers that must be as robust as possible since they must be always available (day and night) to accept students' works. To increase their robustness, they do not depend on the presence of the central databases. They may be deployed within firewalled networks, and maybe polled with encrypted tunnels towards the grading machinery thus ensuring robust security.

Storage servers *s* only serve reports i.e., static anonymous XML files. These servers do not need to access the central databases therefore they are robust: students may then get feedback whenever they want (day and night). To preserve anonymity, reports may only be fetched via non guessable md5-based urls. Storage servers may be implemented with Amazon S3 technology. In our experience, reports from storage servers are often accessed via portals that skin the reports with the university's web conventions.

Teachers prescribe sets of exercises to students and may access their resulting reports. Teachers may also collect students' works (jobs) by any convenient mean and submit batches of jobs for grading. Teachers have (authenticated) access to grades and names through the *t* server.

The author of an exercise also has access to the anonymous reports produced by his exercise in order to improve it.

Information confinement is strict: grading slaves grade anonymous files, storage servers store anonymous files. Among all servers, only grading servers *gd* and *t* (see below) are aware of students' identities.

Servers are customised for one task only. This offers numerous advantages — redundancy to cope with fault tolerancy, — concurrency to increase throughput, — versatility where a new version of a component may be plugged into the architecture and compared (in speed, resource consumption, accuracy) with its previous version without disrupting the entire infrastructure.

For security reasons, the grading machinery is (quite often) not publicly nor directly reachable. The grading slaves are virtual machines run with QEMU, VMware, VirtualBox, etc. as explained in Section 3.2.

3.1 Exercise format

An exercise is a tar gzipped file (not dissimilar to a Java `jar` file or an OpenDocument file) containing an XML descriptor and all the files required to operate the exercise. This central descriptor rules the various aspects of the entire life-cycle of the exercise, successively: autocheck, publish, download, install, grade. These phases are shortly explained hereafter in increasing complexity order.

download The descriptor specifies the stem and the files required by the student in order to practise the exercise. The descriptor also describes for each question, the files or directories expected from the student and may contain additional hints such as a template to fill for an answer or the size of the expected work. In the case of an exercise with a single question expecting a single file, an FW4EX-compliant client (see Section 3.4) may use these hints to display an appropriate widget to capture interactively an answer on one or multiple lines.

grade The descriptor defines which scripts (written in whatever scripting language) are used to grade student's files and how these scripts may be combined. It also specifies which kind of grading slave is required that is, which OS (Debian, Windows, etc), which CPU, etc. The descriptor also mentions CPU duration limits, output limits, etc.

install The descriptor defines how an exercise is installed on a grading slave (usually a virtual machine) or on a student's computer. This installation may require to uncompress data files, to precompile libraries, etc. The installation on a grading slave is different from the installation on the student's computer since grading scripts are not disclosed to the student nor they are communicated to the student's computer.

autocheck The descriptor defines a number of "pseudo jobs" paired with the expected grade they should get. This allows — to check whether the grading slave contains all the software required to grade a specific exercise, — to ensure non regression when an author evolves an exercise.

Most often it is advised to have at least three pseudo jobs: the null one that contains nothing and expects to obtain a 0/20 grade. The perfect one that should be graded 20/20 and an intermediate one that checks that in-between grades are indeed possible!

publish When installed and autochecked, an exercise is ready to grade real students works. It is then publicly available.

The descriptor also defines some meta-data to characterise the exercise: name, requirements, summary, tags, etc. These are needed by *e* servers to inform students and teachers about the exercises that are ready to be selected.

The XML descriptor is the file that ties all the information required to operate the exercise through all the steps of its life-cycle. The exercise file is autonomous and, similarly to a WebApp, it may be deployed or migrated from one application server to some others.

Stems, reports and other information coming out of the grading platform are XML documents therefore, they are skinnable via XSLT style sheets to fit university look and feel. They may also be processed to fill scholar databases, grade books or any other tool that needs information about grading. These normalized XML documents isolate the grading infrastructure from local idiosyncrasies.

3.2 Confinement

Our experience shows us that students make mistakes (infinite loops, deadlocks, etc.). Their programs need to be confined in time and cpu consumption. Their production must also be limited: not much than a given number of bytes written to files or on output streams. Though rare, some students' programs are malicious and must be tightly restricted (are they allowed to post mails, to open sockets, to start background processes, to destroy resources to prevent grading other student's files, etc.) For all these reasons, we run students' programs under students' accounts (so they do not harm teachers' files) within virtual machines (QEMU or VMware) and pay much attention to offer the same initial conditions for all jobs.

To use virtual machines also guarantees to be able to reproduce for ever the same conditions in which an exercise should be graded. We previously had bad experiences with updates breaking configuration or runtime of some languages. Virtual machines also allows to offer various Operating Systems (Debian, Windows, etc.), CPUs or networked machines for client-server programs (for instance in PHP).

Our experience also shows us that authors make mistakes. Their programs need to be similarly confined otherwise they may bog down servers, destroy common resources and harm student's experience. To grade an exercise is therefore a complex task where student's errors should appear in student's report, author's errors should appear in author's reports and FW4EX errors should be routed to FW4EX maintainers.

3.3 Grading scripts

Scripts are regular programs written in whatever language fits the task. They are run within the directory where student's files are deployed, they may read or run author's owned files (stored elsewhere) as well as read or run FW4EX libraries (stored elsewhere).

Grading may be performed by comparison (with teachers' solutions) or by verification (checking whether some property is obtained). Grading by comparison requires the author of the exercise to write his own solution but allows to randomly generate input data.

The exercise descriptor also specifies how to combine the various scripts that is, stop at first error, ignore some errors, etc.

Support libraries help writing very concise grading scripts — for specific languages (for instance in Java, we refine the JUnit framework to count the number of successful/failed assertions in addition to the number of successful/failed tests) or — for specific situations (for instance in bash, input data may be given as command options, input streams, data file names, etc.).

3.4 FW4EX-compliant clients

We strongly believe that, apart for introductory courses to Computer Science, programming should be done with professional tools such as usual programming environment (IDE). Browsers, applets cannot be considered as decent substitutes for real IDE. An FW4EX-compliant client is an IDE (often an IDE plug-in) able to speak to the three types of public servers (*e*, *a* and *s*) according to some REST protocols (Fielding and Taylor, 2002).

These protocols allow a student to authenticate, to obtain stem, to submit work and to, finally, get a

report. When submitting a job and following REST principles, the client receives the URL where the report will pop up on some *s* server.

From an author's point of view, it is similarly possible to submit an exercise, get the corresponding autocheck report leading to the grading reports of the pseudo-jobs. Additional services exist for FW4EX maintainers to manage jobs, exercises, configuration.

Presently we have three FW4EX-compliant clients. The first is an AJAX FW4EX-compliant client running in any regular browser. It fulfills all of the needs but for the installation of the exercise on the student's computer which cannot be fully automated for security reason. It manages the display to accommodate various types of exercises (see experiments in Section 4): one-liners are captured via a one-line text widget, short scripts are captured via a textarea widget, multiple files are captured via a file upload widget.

Since it mostly handles XML documents, the AJAX FW4EX-compliant client imposes its own XSLT style sheets to display them with the wished look and feel. Additional treatments may also be performed such as using a tree widget to iconize series of lengthy tests in the grading report. The report is therefore largely independent of the display technology.

The AJAX FW4EX-compliant client may be embedded in a university portal.

Two other clients were developped as proofs of concepts: one for Eclipse (for Java, C, php, etc.) and one for Scite[6]. In these IDEs, similarly to what we did for DrScheme (Brygoo et al., 2002), a single button or menu will manage authentication, show stem, install accompanying files, pack student's work, submit it and display the associated report, if available, as annotations to source code.

4 Experiments with FW4EX

We deployed the FW4EX platform in September 2008. Two experiments were conducted during the two teaching semesters, a third one took place at the end of January 2009 with quite a different modality.

4.1 One-liners

For a course on Posix skill (mainly shell and Makefile) we deploy approximately 40 exercises. These exercises are "one liners" since they contain a single question that asks for options (for utilities such as `tr`, `sort`, `sed`, etc.) or for whole commands for various tasks (sifting, sorting, regexp-ing, etc.). Stems are terse and imprecise (but always give an example of use) in order to let students think about limit cases such as empty streams, empty files, files out of the current directory, files with weird names, etc. The goal of these formative exercises was to encourage students to read the associated man pages in order to be familiar with the 2 to 4 most useful options with each of these utilities.

These exercises are permanently available for all students, students are not limited in number of submissions. Reports are very detailed, every test is explained, results are shown and reasons for success or failure are verbalised. Students have to analyse these reports and sometimes discover that some limit cases are indeed possible with respect to the stem. In the actual deployment, grading reports are obtained after 20 seconds. Such a duration compels students to consider that the grader is not a fast debugging tool so debug should be done appropriately on their computer before asking to be graded.

We offer roughly 5 exercises per week for 7 weeks. We start from exercises asking for options and end with exercises asking for small shell scripts (less than 10 lines). With help of the support libraries, grading scripts were reduced to a line or two.

On the 120 students enrolled for the course, half of them tried at least one exercise. The platform had been serving (night and day) around 3500 submissions per semester.

The exercises were not prescribed, only voluntary students try them. As already reported (Woit and Mason, 2003), students did not volunteer easily. Apart the initial curiosity of the first week, we only noted the usual peak during the week that precede examinations.

4.2 Examinations

In the same course, we use the FW4EX platform to grade the mid-term examination and the two final examinations. The examinations contain three summative exercises each containing 1 to 6 rather independent questions. Contrarily to the one-liners, examinations are very carefully specified. Stems are precise, they ask for scripts (or Makefile rules) performing various tasks. Examples are always given, deliverables are specified and some hints are given about how grading will be performed.

These examinations last 3 hours. As usual the mid-term examination mainly serves to prepare the students for the final examinations. The student's works are sent to the grader after the examination

[6] www.scintilla.org

therefore students do no have any feedback during the examination.

In this setting, we had 120 sets of students' files to grade. The exercise file (containing the examination) was prepared with 6 pseudo jobs. We grade the whole set of students' files 4 times in order to tune the grader: we had to cope with misnamed scripts or files, to fix an encoding problem (UTF8 versus Latin1) and to slightly alter the weight of two questions. Approximately 60 seconds were necessary to grade one student.

The design and test of these examinations (writing the stem, writing a solution, writing the grading scripts and testing them over pseudo jobs) around two days of work.

Five days were necessary for the examinations before the reports and grades were made available to the students (a student may only access his own report). Among these five days, three were alloted to the teaching assistants to check whether the results were agreeing with the behaviour of the students during lab sessions.

The mid-term examination was also made available as a regular (although huge) exercise itself. Students may then retry this examination in a less stressed context (at home) but, at the same time, be graded by the exact grader used at the examination. This is what we call "dynamic annals". Only 6 students retried the examination, 1 among them got a grade greater than 19/20 in 4 attempts, the others topped at 6/20 with 1 to 6 attempts before abandon.

The examination was held on the computers of the laboratories where no plagiarism was possible. Plagiarism detection will be considered as an option for batch grading.

4.3 Programming contest

The "Journée Francilienne de Programmation" is a programming contest for undergraduate students from various Parisian universities. A dozen of programming languages are indeed possible to solve the proposed problem. The contest lasts 5 hours during which 12 teams of voluntary students regularly submit their work (120 uploads) and get their grade (the average grading time was 45 seconds) in order to appreciate how they perform with respect to the other teams. Besides individual grading, the portal runs additional scripts using the reports from FW4EX to deliver bonus points to teams' works (mainly based on the speed of programs).

The next edition of this contest will use the same grading infrastructure.

4.4 Summary of experiments

These three experiments show the versatility of the FW4EX platform. One-liners are multiply submitted and graded whereas examinations are just graded once. Examinations and contests are read offline by humans but one-liners are just automatically graded. One-liners are graded in a couple of seconds, examinations and contests take a minute.

5 Related work

While there are many mechanised graders on the market, there are often tailored to a single programming language or tightly bound to scholar databases or imposing a precise workflow. All this characteristics make difficult to re-use these embedded graders.

The BOSS Online Submission System[7] (Joy et al., 2005) is a course management tool. It allows students to submit assignments online securely, and contains a selection of tools to allow staff to mark assignments online and to manage their modules efficiently. Plagiarism detection software is included. Interaction with BOSS may use an application or a web front end.

BOSS is a complete course-oriented web application whereas FW4EX is only a set of protocols to operate a grading infrastructure. On the other hand FW4EX achieves better robustness against malicious submissions and scalability. However due to the flexible architecture of BOSS that uses factories to hide implementation choices or binding constraints with external resources, it would not be difficult to let BOSS use FW4EX.

CourseMaker[8] (Higgins et al., 2005) is a web-based, easy-to-use course creation package commercialised by the Connect company. It is now roughly similar in functionalities to Blackboard or Sakai. Like BOSS, it contains a number of tools – to check typography, syntax, comments, – and to detect plagiarism. CourseMaker is a complete solution that contains courses documents, exchanges information with scholar databases, deploys student clients application or web-based forms to collect work. CourseMaker hosts, on its proper network, courses along with their exercises as QTI[9] files.

There again, FW4EX is different. While CourseMaker is course-oriented, FW4EX is exercise-centric.

ASAP (Douce et al., 2005b) is a new initiative that shares a number of goals with FW4EX. ASAP is com-

[7] http://www.dcs.warwick.ac.uk/boss/

[8] www.coursemaker.co.uk

[9] www.imsglobal.org

ponentized with respect to the JISC (Joint Information Systems Committee) e-learning framework. It separates grading from submission and it also uses XML documents as a way to transmit information between components. FW4EX improves on ASAP with its emphasis on protocols, its robustness with the use of virtual machines and its attention to the whole lifecycle of exercises.

6 Conclusion

In this paper, we present the FW4EX project, an attempt to build a grading infrastructure that can be put to work into various learning environments. A set of REST-based protocols allows these systems or IDEs to operate the grader. A "standard" descriptor and a file format specification are proposed to reduce the deployment of new exercises to a simple file copy. We finally describe some experiments that illustrate the neutrality and versatility of the associated platform: one-liners, full examinations, programming contests in multiple programming languages.

We believe that this project provides a strong basis to foster an eco-system for mechanised grading thus allowing teachers to focus on the sole exercises and not on the grading infrastructure. Yes, this is more work for teachers to imagine a stem, write (and test) a solution, design some grading scripts but, once done, this may last for ever!

More information on the FW4EX project is available on the site `paracamplus.org`.

REFERENCES

Brygoo, A., Durand, T., Manoury, P., Queinnec, C., and Soria, M. (2002). Experiment around a training engine. In *IFIP WCC 2002 – World Computer Congress*, Montréal (Canada). IFIP.

Daly, C. and Waldron, J. (2004). Assessing the assessment of programming ability. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 210–213, New York, NY, USA. ACM.

Douce, C., Livingstone, D., and Orwell, J. (2005a). Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3):4.

Douce, C., Livingstone, D., Orwell, J., Grindle, S., and Cobb, J. (2005b). A technical perspective on asap - automated system for assessment of programming. In *Proceedings of the 9th CAA Conference, Loughborough University*.

Ellsworth, C. C., James B. Fenwick, J., and Kurtz, B. L. (2004). The quiver system. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 205–209, New York, NY, USA. ACM.

Fielding, R. T. and Taylor, R. N. (2002). Principled design of the modern web architecture. *ACM Trans. Interet Technol.*, 2(2):115–150.

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., and Felleisen, M. (2002). Drscheme: A programming environment for scheme. *Journal of Functional Programming*, 12:369–388.

Fu, X., Peltsverger, B., Qian, K., Tao, L., and Liu, J. (2008). Apogee: automated project grading and instant feedback system for web based computing. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 77–81, New York, NY, USA. ACM.

Higgins, C. A., Gray, G., Symeonidis, P., and Tsintsifas, A. (2005). Automated assessment and experiences of teaching programming. *J. Educ. Resour. Comput.*, 5(3):5.

Hill, C., Slator, B. M., and Daniels, L. M. (2005). The grader in programmingland. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 211–215, New York, NY, USA. ACM.

Joy, M., Griffiths, N., and Boyatt, R. (2005). The boss online submission and assessment system. *J. Educ. Resour. Comput.*, 5(3):2.

Noonan, R. E. (2006). The back end of a grading system. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 56–60, New York, NY, USA. ACM.

Queinnec, C. and Chailloux, E. (2002). Une expérience de notation en masse. In *TICE 2002 – Technologies de l'Information et de la Communication dans les Enseignements d'Ingénieurs et dans l'industrie – Conférences ateliers*, pages 403–404, Lyon (France). Institut National des Sciences Appliquées de Lyon. version complète disponible en `http://lip6.fr/Christian.Queinnec/PDF/cfsreport.pdf`.

Woit, D. and Mason, D. (2003). Effectiveness of online assessment. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 137–141, New York, NY, USA. ACM.