Model-based Student Admission
(Position Paper)

Vadim Zaytsev

Institute of Informatics,
Universiteit van Amsterdam,
The Netherlands

Abstract. We should test people the same way we test software.

1 Motivation

There are many established ways to test software, most of them could be boiled down to forming expectations about the behaviour of a system under test, encapsulating them in a model and exploiting the model to test the actual system — by proving properties over it, generating test data from it, and in general confirming that the behaviour observed from the system is consistent with the behaviour expected by the model. The discussion topic we would like to raise with this paper is the possibility to test students the same way.

In the case study included in the remainder of this short paper, we present a Master programme in software engineering and explain its admission procedure, which involves examining self-studying students. We also notice how assessment based on learning objectives is similar to testing a software system based on its model, and propose an infrastructure that is useful at least under our particular circumstances, and perhaps in a much wider context as well.

2 Software Engineering in Amsterdam

Master of Science in Software Engineering[1] is an intensive one year programme of graduate education at the University of Amsterdam. It graduates around 60 students yearly and beside the final project contains six courses for 6 ECTS each:

- *Software Architecture* (high level system design and system modelling)
- *Software Specification and Testing* (type systems, testing, verification)
- *Requirements Engineering* (elicitation, analysis and negotiation)
- *Software Evolution* (metaprogramming and static analysis of source code)
- *Software Process* (management, integration, deployment, maintenance)
- *Software Construction* (MDE, design patterns and programming styles)
- *Preparation Master Project* (experimentation, reading, writing, planning)

The final project itself (18 ECTS) involves a replication study of a practically applicable academic paper and takes upward of four months.

[1] `http://www.software-engineering-amsterdam.nl`

3 Intake procedure

Admission to the programme is based on having a Bachelor degree or its equivalent, as well as on demonstrating sufficient levels in skills like software development, logic, term rewriting, compiler construction and algorithmics. At the entry level, students are expected to be able to manipulate simple mathematical formulae and have some basic knowledge of (semi-)automated testing techniques, have some experience in constructing, versioning, packaging and maintaining software. Given the admission guidelines and the Dutch environment around the University of Amsterdam, we have at least these five large categories of applicants well represented:

- **University students** following the classic educational road by obtaining a Bachelor degree from the same or a neighbouring university, in computer science, software engineering, business computer science, etc., and moving on to a (allegedly more practice-oriented) Master programme.
- **Students switching** their focus from another programme: mathematics, artificial intelligence, computational science or bioinformatics, for either growing to dislike their original choice or seeking some fresh technical topics.
- **International students** from all over the world with degrees administratively equivalent to the ones from the first two groups, but having occasionally followed a very different curriculum.
- **High school students** which have also received a Bachelor diploma (a typical setup in the Netherlands) yet followed a considerably more technical curriculum without much attention, if any at all, to formal methods or any other underlying theory for that matter.
- **Software engineers** who obtained their Bachelor degree in the past (sometimes decades ago) and have been working as practitioners of software engineering ever since, until they decided out of curiosity or as a strategy to boost their CV, to relearn their basics at a more advanced and modern level.

4 Premaster courses

Each intake procedure is done individually. It resembles a job interview, occasionally involves a formal test and is performed after receiving, processing and inspecting the application which includes a letter of motivation. Within approximately an hour, a programme coordinator assesses the skills of the applicant and converges to a verdict which can be in a form of rejection, acceptance, conditional acceptance or redirection to a different programme.

Conditional acceptance is the most interesting outcome, because it means that one or more premaster courses are assigned to the student. Each premaster course typically involve many hours of self study, occasional interactive supervision and finally an examination.

An example of a premaster course is *Mathematical Logic* which is meant for technically strong applicants with a weak or absent formal background. The learning objectives revolve around quantifiers, induction proofs, propositional logic, syllogistic reasoning, grammars and automata — all on a very basic level, just enough for potential students to not struggle when they see a formula or

discuss infinitely large objects. The material is not limited: we advise the students to use *Logic in Action* [4], Wikipedia articles, Coursera courses, Khan Academy material and any other resources googleable from the internet or borrowable from the local library.

Another example is *Compiler Construction* which is assigned to applicants strong in modern software engineering techniques such as web app development with no prior exposure to system software and/or DSL/MDE. A student involved in this course is advised to get acquainted with the first chapters of the Dragon Book [1] and expected to be able to list the components of a typical compiler backend, assess their usefulness and role in performing a particular software language engineering task. Many students also turn to other books, YouTube videos, Coursera courses, etc.

5 Model-based admission

Consider the premaster course on logic briefly introduced above. Given that we can rely on technical knowledge of our potential students, we ultimately want them to see a connection between familiar activities and their formal methods counterparts. This is the case for different topics within the premaster material:

$$\forall x,\ x \in A \wedge x \in B \Rightarrow x \in C \qquad \Longleftrightarrow \qquad A \cap B \subseteq C$$

A quantifier-based formula on the left is universally equivalent to a set-based formula on the right, and there are many exercises in our examinations to either recognise such pairs or infer one if given the other. However, this is also the case for bridges between other areas and formal methods: e.g., substitution in a mathematical formula has always been a somewhat difficult topic, unless explained in terms of programming (bounded variables are local variables, unbounded are global, and thus it is much easier for practising programmers to come up with the idea of renaming in case of clashes on their own). Cardinality estimation skills, especially for infinite ($\aleph_0$ or $\aleph_1$) sets, is also usually assessed based on questions about the number of possible programs in a language, or chess moves, or sorting algorithms of particular structure.

Hence, we can systematically list these equivalence relations that we expect a student to confirm, and generate enough exercises (according to time constraints) to test them. Such exercises can be checked automatically. Furthermore, we can also make solid claims about coverage of the studied material by the assignments, both generated and answered correctly. Figure 1 shows a simple example.

Hence, we achieve drastic decreases of time spent on preparing the examination assignments (they are generated), better examination service in general (assignments are individual and are not reused through the years), less time spent on assessment (automated grading) and better learning analytics (extensive use of coverage criteria can also eventually lead to developing an interactive system that keeps exploring knowledge areas and localising white spots in order to make the fairest estimation possible of the skills of the student under test). The obvious threats revolve around the creation effort, the quality level and the validation of the specifications.

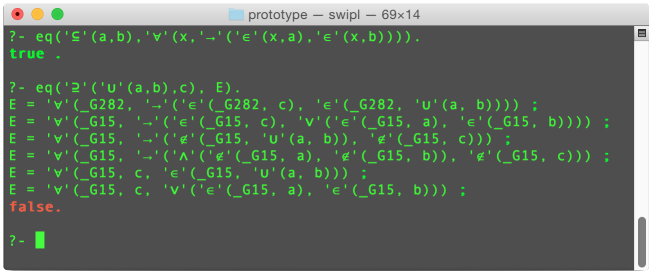

Fig. 1. An example of checking an equivalence of two formulae (top) and generating possible formulae equivalent to a given one (bottom). The program behind both is a trivial universal declarative model of around a dozen lines of Prolog code. `http://gist.github.com/grammarware/813374043858030b2059`.

6 Conclusion

From the constructive alignment point of view [2], premaster courses are unique in a sense that they have loosely defined learning objectives, a virtually nonexisting or at least unknown set of educational tasks and an absolutely crucial assessment procedure. We propose a setup where an equivalence model is formed around the learning objectives in such a way that a set of test cases is generated to be used as an input (i.e., exam) for the system under test (i.e., student). The automation enabled by this approach allows us to provide better testing services (i.e., fair admission) with less performance sacrifices (i.e., teacher time). In recent decades we have become good in software testing, so we should really test people the same way we test software.

Many courses start with either a formal quiz or a informal collection of data about new students' starting level of knowledge — this is, in fact, known as acceptance testing in software engineering. Programming exercised were traditionally checked by differential testing [3]. Weekly assignments are usually composed as unit tests, exercising one particular topic, while midterm and final examinations are used as a regression testing tool. Final assessment is also a form of conformance testing with respect to claimed exit qualifications. Maybe, indeed, we should exploit these similarities, align software testing methodologies with student assessment practices and borrow useful experience both ways?

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.
2. J. Biggs and C. Tang. *Teaching for Quality Learning at University*. McGraw-Hill and Open University Press, 2011.
3. C. Douce, D. Livingstone, and J. Orwell. Automatic Test-Based Assessment of Programming: A Review. *Journal of Educational Resources in Computing*, 5(3), Sept. 2005.
4. Open Course Project. Logic in Action. `http://www.logicinaction.org/`.