Moulinog

Yernaux, Gonzague; Vanhoof, Wim; Schumacher, Laurent

Published in:
Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming, PPDP 2020 - Part of BOPL 2020 - Bologna Federated Conference on Programming Languages 2020

DOI:
10.1145/3414080.3414100
10.1145/3414080.3414100

Publication date:
2020

Document Version
Peer reviewed version

Moulinog: A Generator of Random Student Assignments Written in Prolog

Gonzague Yernaux
Wim Vanhoof
Laurent Schumacher
{gonzague.yernaux,wim.vanhoof,laurent.schumacher}@unamur.be
University of Namur
Faculty of Computer Science
Namur, Belgium

ABSTRACT

We introduce, describe and discuss the potentialities of Moulinog, a tool created during the COVID-19 lockdown, designed to generate individual questionnaires for the remote evaluation of large classrooms. Starting with a list of students and a series of predicates constituting a pool of parametric questions along with rules for their parametrization, Mouling generates a list of individual questionnaires, together with a shell script allowing an easy emailing of the (password-protected) questionnaires to the students. The tool's use in practice is illustrated on a particular course case for which it has proven to be both useful and time-saving.

CCS CONCEPTS

• **Applied computing** → **Computer-assisted instruction**; • **Theory of computation** → **Constraint and logic programming**; *Generating random combinatorial structures.*

KEYWORDS

Remote education, Prolog, Pedagogic tool, Assignment generation

ACM Reference Format:
Gonzague Yernaux, Wim Vanhoof, and Laurent Schumacher. 2020. Moulinog: A Generator of Random Student Assignments Written in Prolog. In *22nd International Symposium on Principles and Practice of Declarative Programming (PPDP '20), September 8–10, 2020, Bologna, Italy.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3414080.3414100

1 INTRODUCTION

The year 2020 appears to be definitely marked by its sanitary crisis caused by the COVID-19 worldwide spread. Although more and more positively under control, the virus still affects many institutions at the time of writing, amongst which teaching establishments. One of the challenges encountered by many schools and universities is indeed to maximize the remote continuation of academic duties. Often with not enough time – or desire – to design active anti-cheating techniques, some academic structures must look into ways to measure the students' understanding of concepts remotely.

In this experience report, we introduce Moulinog[1], a Prolog-based tool generating individual exam papers from a corpus of exam question templates and associated parametrization rules. The Moulinog core, which is used to generate exam copies for several courses at the University of Namur, allows two levels of customization to generate course-specific questions: first, the knowledge base representing the questions themselves along with their parameters; second, rules over these parameters that allow to generate specific exercise statements obeying the laws of the covered course material – all this by taking advantage of Prolog's operational peculiarities.

Other pandemic-related educational approaches exist all over the world, often focused on broader concerns such as risk management and teaching continuation in general [7], or the continuation of education in particular study domains [4, 5] or contexts [2]. In this work, rather than teaching in general we try and focus on the issue of having students take certificative tests in establishments usually resorting to face-to-face evaluation, and as such in need to adapt quickly and efficiently to administer written exams remotely. Although many existing technologies are available for this task, we choose to exploit the mechanisms of Prolog for several reasons that are outlined in the paper. To our knowledge, such an approach, aiming both to solve the current practical examination issues and to do so with a declarative set of instructions, has not been explored yet. Moreover, the tool presented here being generic and open-source, we believe that it can constitute fertile ground for (adapted) use in teaching environments as well as an interesting base for fundamental reflections.

The present report is structured as follows: in Section 2 we present more of Moulinog by describing its inputs and outputs in a typical university workflow, and we further dwell into the two levels of customization offered by Moulinog. In Section 3 we discuss the tool's use in practice, which we illustrate on the case of a computer architecture course held at the University of Namur (Belgium). Then, in Section 4 we discuss the advantages of such a solution, and in particular the benefits of having such a tool written in Prolog, before concluding with some final remarks and pointers for future improvements.

PPDP '20, September 8–10, 2020, Bologna, Italy

ACM ISBN 978-1-4503-8821-4/20/09…$15.00
https://doi.org/10.1145/3414080.3414100

[1]The first fully functional and open-source version of Moulinog is accessible at the following address: https://github.com/Gounzy/Moulinog.

2 TOOL OVERVIEW

2.1 Inputs and outputs

Moulinog is designed to be used for generating an arbitrary number of questionnaires. A typical use, be it in a university or a high school, would involve (1) an automatic attribution of the questions to the students (based e.g. on a list of students), (2) some kind of automatic mailing processing to generate final files for the students (in a portable format such as PDF), (3) possibly some kind of (e.g. password-based) protection ensuring that the files are only available for reading at a certain moment in time (e.g. when the password is revealed), and (4) an automatic distribution of the files to the students (e.g. through an emailing command). To fit in this workflow Moulinog receives as input a list of students (with names and email addresses) and outputs a .csv file processable by a mailing processing software, in order to generate the final files that can be sent through a shell script. These steps are represented on Figure 1. Observe Moulinog's central role in automatizing several steps of the process: the questions generation, but also the shell script that will effectively send the questionnaires to the students. Except for the csv-to-pdf operation, which is typically better achieved by a standard text processing software, Moulinog automatizes the key steps of the process.

2.2 Questions and parameters

The only information needed by Moulinog in order to generate random question sheets are the questions themselves, as well as information about their possible variations (i.e. their parametrization). The following code constitutes an example question from a computer architecture course, along with some of its parameters.

```
question(cache, ["Let us consider a ", type, "
    cache memory, using ", updatestrat, " update
    strategy, with ", nbwords, " per memory block
    and ", nbbytes, " per word. Supposing that
    there are ", totalwords, "words in the cache
    in total and that the addresses are 8-bit each
    , represent the cache's structure and execute
    the following sequence of read/write
    instructions: ", sequence, ". Explain how the
    cache's contents evolves through the execution
    ."]).
question(cache, ["Explain how a direct mapped
    cache works."]).
param(cache\type, "direct mapped").
param(cache\type, "fully associative").
param(cache\type, "2-way set associative").
param(cache\updatestrat, "write-back").
param(cache\updatestrat, "write-through").
% ...
```

In what follows, we use the usual concise notation p/n to refer to a predicate built upon the predicate symbol p and having arity n.

The first argument of *question*/2 indicates the question's category identifier, which allows to formulate different potential questions belonging to the same category of questions (e.g. all the questions related to a particular lesson). As for the non-string atoms appearing as list elements in the second argument of *question*/2, these indicate parametric positions in the question's formulation. Moulinog will then fill these positions with instantiated parameters. The possible instantiations of a parameter with identifier id_p in a question with identifier id_q are indicated in those clauses of *param*/2 of which the first argument unifies with $id_q \backslash id_p$.

In order to generate different questions or exercises (in the following, we use both terminologies indifferently) based on a knowledge base composed of definitions for *question*/2 and *param*/2, the program creates lists of different questions, the uniqueness of each generated question being guaranteed either by the fact that it is built upon another *question*/2 clause, or because of its unique parametrization. The question generation itself is handled by the following predicate, that can be considered as Moulinog's core operation:

```
questions(CategoryId, Questions):-
  findall(Q, question(CategoryId, Q),
      QuestionsPlain),
  zip_questions_with_params(QuestionsPlain,
      QuestionsWithParams),
  parametrize(QuestionsWithParams, Questions).
```

Note the use of *findall*/3 dangerously allowing the number of Prolog computations to explode. The choice was indeed made to keep the high-level approach generic while allowing each instructor to customize his or her questions as he/she pleases, hence this potentially very broad search process in Moulinog's core. This particular insight will further be discussed in Section 4.

Note that *zip_questions_with_params*/2 is a helper predicate that fetches the identifiers of all parameters appearing in the questions. As for the *parametrize*/2 operation, it also makes use of *findall*/3 to get all possible parametrizations of each question, for the same reason as *questions*/2 does. In the process, a randomization operation makes sure that the students are all given different instantiations of questions for each category, at least if such an attribution is possible. If, however, all the parametrized questions have been distributed for a category, Moulinog reassembles the whole pool of questions in that category and resumes the random drawing for the students that are next in line, after having warned the user of this overflow possibly causing multiple students to get the same questions for that category. In most cases the risk of having two students that have more than two identical questions remains relatively low, given that all the question-picking operations are achieved based on a random draw. It is furthermore unlikely that several category pools happen to be exhausted simultaneously. Still it is the instructor's choice to either ensure an entirely distributed assignment of the questions (by writing enough exercise and/or parameter variations) or allow (partially) duplicated questionnaires in the classroom.

2.3 Constraints

Questions can be parametrized with simple text variations as illustrated above, or thanks to the use of more complex rules called *constraints*. The constraints are a way of taking advantage of Prolog's deduction mechanisms to generate more elaborate questions that fit specific rules. The following code, for instance, shows a question for which the parameter must be filled with a combination of bits that will allow the logical function that it represents to

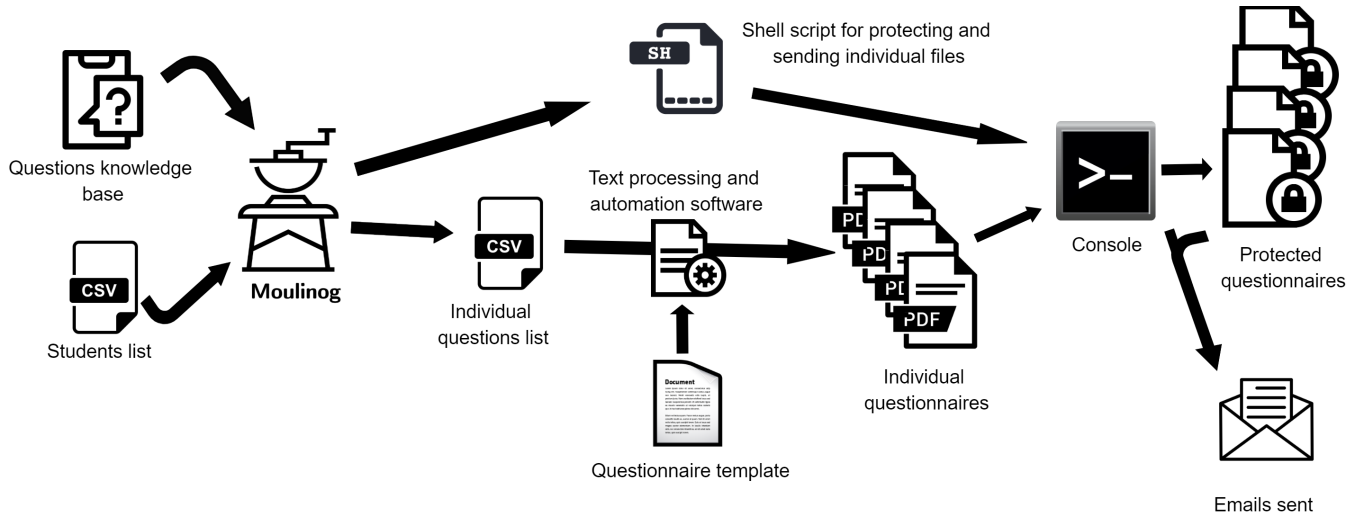

Figure 1: A complete workflow for randomly attributing student assignments

be, at least to some extent, simplifiable with the well-known Karnaugh map method [3]. Making use of Prolog's in/out modes this way saves the instructor from having to imagine many (more than one hundred in the case studied in the following section) different configurations of binary functions.

```
question(karnaugh, ["Let us consider a binary
    function F(A,B,C,D) that outputs 1 if and only
     if it is given the following inputs: ",
    inputs, ". \r\nEstablish the truth table for F
    , derive its Karnaugh map and, finally, draw
    its minimal logic cicuit."]).
param(karnaugh\inputs, X):-
  binary_function(Values),
  possible_karnaugh(Values),
  values_to_string(Values, X).
binary_function([]).
binary_function([f(A,B,C,D)|Values]):-
  A is random(2), B is random(2), C is random(2),
      D is random(2),
  binary_function(Values).
possible_karnaugh(Values):-
  select(Value1, Values, OtherValues),
  member(Value2, OtherValues),
  one_diff(Value1, Value2).
one_diff(f(A1,B1,C1,_), f(A1,B1,C1,_)).
one_diff(f(A1,B1,_,D1), f(A1,B1,_,D1)).
one_diff(f(A1,_,C1,D1), f(A1,_,C1,D1)).
one_diff(f(_,B1,C1,D1), f(_,B1,C1,D1)).
% ...
```

In the above code, the parameter used in the *question*/2 clause is defined as a binary function (being a set of 4-uples from $\{0,1\}^4$) that is simplifiable by Karnaugh's method (which is assessed in the *possible_karnaugh*/1 predicate).

3 CASE STUDY: A COURSE OF COMPUTER ARCHITECTURE

The code portions used as examples above are selected (and translated) parts of Moulinog's parametrization for a course of computer architecture given at the university of Namur (Belgium). The course usually ends with a written exam of five exercises, each being related to a particular lesson. The course had to address the following constraints:

- The terms of the evaluation had to be adapted so as to be achieved remotely, while staying as close as possible to the initially announced modalities (written exam, five questions, four hours).
- No active anti-cheating measures could be taken (such as e.g. webcam surveillance); but the cheating possibilities had to be limited.
- All the 150+ students have to pass the test at the same time.
- The test cannot rely on the university servers for the whole time of the exam. In fact, accesses to the servers have to be kept to a minimum, to avoid connection hazards as much as possible.
- The correction of the test should not be proportionally time consuming with the number of different questionnaires; in other word, correctors should not have too much additional work than with a unique questionnaire.

The following workflow was then followed for this course:

(1) Get a list of students with their names and email addresses.
(2) Create a pool of parametric questions for each category.
(3) Add constraints over some of the parameters to ensure the output's fitness towards the learning outcomes (e.g. Karnaugh maps, consistent types of cache memory structures, different yet consistent assembly code to analyse or produce).
(4) Execute Moulinog, yielding a shell script and a file containing the individualized exercises.

(5) Create a template document containing the exam instructions and structure.
(6) Thanks to a mailing software, fill in the blanks of the template document with the informations of the listing (name of the student and parametrized questions), yielding one PDF per student, saved as <STUDENT_NAME.pdf>.
(7) Execute the shell script in the folder containing all the PDF documents, thereby enciphering each document with an encryption algorithm and sending each PDF to its intended recipient.
(8) At the beginning of the online test, publish the password allowing to decipher the documents. Each student must then produce and send (pictures of) his/her answer sheet on a given institutional server.

The calibrating of the *question*/2 variations was done by the instructors in such a way that two instances of a question were sufficiently different to prevent the students from easy cheating with each other's help, while being equally difficult and relevant to the course material to which it is related. Calibrating the questions this way was a success in this use case (namely thanks to the many examples of past exams where each time the same competences were expected) but definitely constitutes one of the challenges of writing questions. In fact, it is the same challenge that an instructor usually faces when having to write different questionnaires for different exam sessions of the same year.

Moreover, the questionnaires were designed in such a way that there were differences between the students assessments, but also such that the same core skills were required, so that the correctors easily knew what they were looking for on the answer sheets. Having a look at the questionnaire before correcting a student's answers was often enough to grasp the specificities that this particular student had to take into account in his answers. This is again in relation with the calibration of the questions, which proved to be central to a Moulinog use in practice.

This process fulfilled all the constraints outlined above, by making the invention of an exam as simple as programming a few predicates describing its exercises. Obviously, having to come up with more than 150 different questions is harder than imagining one and the same exam file distributed to all of the students, as is more usual. But the time spent on building parametric questions that are to be fed to Moulinog is well-spent time, given that there are now enough different questionnaires to be used in several years to come (considering that the future exams will be physically attended).

4 DISCUSSION: ADVANTAGES AND CHALLENGES OF A PROLOG-BASED ACADEMIC TOOL

Moulinog is a tool tailored to be quickly and easily usable in the current COVID-19 circumstances. Although particularly useful for remote evaluation practices, the concept of generating individual assignments based on a knowledge base describing not only question templates, but also the whole rationale behind a given course's logic, can readily be used in other cases as well, given that it is a way of facilitating the redaction of questions or exercises statements – one application of which being the visualization of all the possible questions for a given course, not only constituting a database of archive exam questions or exercises for the course, but also actually describing in a measurable way what is concretely expected from students that pass the course. These side insights can e.g. help instructors better dig into their success requirements and refine their learning outcomes or even the way they give their lessons.

The generator is designed to be used by instructors that are not necessarily familiar with Prolog, or any programming language for that matter. Using a declarative language for this is a challenge, but it has some advantages and Prolog has been advocated as an excellent programming [6] and Artificial Intelligence [1] initiation. In this first version of Moulinog, the choice of Prolog as a programming language is also due to its straightforward handling, allowing the pedagogic teams to readily write programs with almost no prior Prolog knowledge (the sole necessary syntactic unit being the clause) manipulating constraints fit for a given application domain, while taking advantage of the powerful inference mechanisms that are crucial to Moulinog. The most basic usage of Moulinog is indeed straightforward: a text editor and a few statements encapsulated in *question*/2 clauses allows for a ready-to-use, almost plug-and-play generation of automatically attributed questions.

More advanced uses (e.g. writing constraints) can be seen as a practical demonstration of the benefits of declarative programming: few code lines producing qualitative results; describing the objectives (through constraints over the question parameters) rather than the algorithms required for the questions generation; but also having to write the constraints in a manner to limit their computational complexity. Indeed, the presence of calls to $findall/3$ in the core predicates of Moulinog puts in the instructor's hand all the responsibility of circumscribing the computations to be carried out: should *all* results be found, or only a subset of them? How to ensure that there are no duplicates in the parameters outputs? These concerns actually match declarative programmers usual worries, while the benefits of using Moulinog are also typically those of any such declarative system. Also note that having these details handled in the writing of the constraints ensures that the tool offers as much flexibility as possible to its users since they are the ones defining the desired computation limits.

5 FINAL REMARKS AND FUTURE WORK

Moulinog's core presented in this report constitutes a minimum viable product of what can become a powerful tool for a pedagogic team: indeed, many potential extension features jump to mind when considering the tool's uses. Some of them are already under development for future exam sessions. Here is a non-exhaustive list of future work to be done in order to let Moulinog become a more complete and accessible tool:

- Adding support for some typical questions customization, e.g. multiple choice questions, true or false statements, etc.
- Adding support for more elaborate parametrization: for instance, allowing constraints to rule over more than one parameter (thereby expressing the link that exists between said parameters), or only allowing certain combinations of parameters in a question (which can for now only be done by writing more than one version of the question). In particular, exploring and exploiting the possibilities of more elaborate

Prolog techniques such as tree-based approaches (e.g. representing a question as a tree with parameters as leaves) or other useful extensions of the language might constitute a promising lead in order to write more complex constraint systems.

- Adding a user interface to list, edit, add and delete questions, parameters and constraints. A simple higher-level interface, possibly coupled with the use of existing tools, can facilitate the use of Moulinog by people not familiar with Prolog or even programming in general while still demonstrating to these people the plus-value of using the Prolog inference system rather than more classical programming approaches.
- Allowing Moulinog to run by itself, independently of a shell script or text processing software. For instance, let students draw lots in the questions pool during a live session.

In our experience, the use of Moulinog seems to have achieved fairness towards other students, fairness towards other years exams and fitness of the test for remote evaluation (based on informal data collection on these levels). This is again thanks to the calibration of Moulinog which output well-balanced questionnaires that were still close enough to the previous years exams examples available to the students. However, a formal and objective satisfaction survey should in our opinion be carried out when Moulinog achieves larger adoption, in order to get statistically enlightening results.

In conclusion, despite its simplicity regarding some matters, Moulinog has successfully been used already, allowing more than 150 students to pass an exam simultaneously and remotely while keeping similar examination conditions as usual. The tool was used by non-regular Prolog users to generate different assignments for each student, so that no two students received the same question. We believe that this is one of the many answers that academic staff can give to the coronavirus disease. Many are indeed looking to keep their business rolling peacefully while taking advantage of the situation, e.g. by integrating innovative techniques into their courses usual workflows. Other courses given at the university of Namur – including a course on computer networks and another on functional and logic programming – are scheduled to use Moulinog as a questionnaire generator and are currently, in that purpose, facing some new specific challenges, which are as many leads for the extension and further enhancement of the system.

REFERENCES

[1] Paul Brna, Alan Bundy, Tony Dodd, Marc Eisenstadt, Chee Looi, Helen Pain, David Robertson, Barbara Smith, and Maarten Someren. 1991. Prolog programming techniques. *Instructional Science* 20 (01 1991), 111–133. https://doi.org/10.1007/BF00120879

[2] Robert Connor Chick, Guy Travis Clifton, Kaitlin M. Peace, Brandon W. Propper, Diane F. Hale, Adnan A. Alseidi, and Timothy J. Vreeland. 2020. Using Technology to Maintain the Education of Residents During the COVID-19 Pandemic. *Journal of Surgical Education* (2020). https://doi.org/10.1016/j.jsurg.2020.03.018

[3] M. Karnaugh. 1953. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics* 72, 5 (1953), 593–599.

[4] Li Li, Qianghong Xv, and Jing Yan. 2020. COVID-19: the need for continuous medical education and training. *The Lancet Respiratory Medicine* 8 (03 2020). https://doi.org/10.1016/S2213-2600(20)30125-9

[5] Eric Liguori and Christoph Winkler. 2020. From Offline to Online: Challenges and Opportunities for Entrepreneurship Education Following the COVID-19 Pandemic. *Entrepreneurship Education and Pedagogy* (03 2020). https://doi.org/10.1177/2515127420916738

[6] David B. Paradice. 1988. Prolog: A Language for Teaching Computer-Based Business Problem Solving. *Journal of Education for Business* 63, 4 (1988), 184–187. https://doi.org/10.1080/08832323.1988.10117306 arXiv:https://doi.org/10.1080/08832323.1988.10117306

[7] Chuanyi Wang, Zhe Cheng, Xiao-Guang Yue, and Michael McAleer. 2020. Risk Management of COVID-19 by Universities in China. *Journal of Risk and Financial Management* 13 (02 2020), 36. https://doi.org/10.3390/jrfm13020036