EdTech WS 06/07: Final Project Report

Generating Parameterized Exercises

Minko Dudev*, Alberto González Palomo†

March 26, 2007

Abstract

In this report we describe the extensions of the eLearning system ActiveMath that we developed in the frame of the Educational Technologies lecture given at Saarland University. We completed a project focused on the generation of parameterized exercises. Our work was reviewed by George Goguadze.

We significantly improved the implementation by adding two different mechanisms: picking random numbers from given intervals (open, closed, continuous and discrete) and composing arbitrarily complex mathematical expressions using an iterative term rewriting algorithm.

Furthermore, our implementation takes into account the user model and provides a set of guarantees for what is picked depending on information from the user model. Also, expressions generated by our algorithm are checked for validity with a Computer Algebra System.

Besides the in-depth description of the implemented changes, we present some problems we encountered and the respective solutions that we found.

1 Background

The eLearning system ActiveMath employs exercise generators to enrich exercises written by content authors. More specifically, exercise generators are implemented as classes that transform the graph of interactions of an exercise in some way in the process of delivery such that the content authors only have to provide minimal amount of information for the system to present the user with new exercises. There is a continuous spectrum of possibilities, from generators that only change small details in manually authored exercises, to generators that need no previous exercise content as everything is generated on the fly. An example of an exercise generator in ActiveMath is the randomizer (org.activemath.exercises.generators.Randomizer). The randomizer is a function that replaces variables in the exercise content by random values to provide entire classes of exercises from a single template.

*mdudev@mpi-inf.mpg.de

†Alberto.Gonzalez@matracas.org, http://www.matracas.org

Figure 1: Pictorial view of the exercise generator function.

2 Problem Description

"Until now, Randomizer was only working with given finite sets of possible values. The current task is aiming to enhance the Randomizer to randomize over any (possibly infinite) intervals, discrete or continuous. Additionally to that, it should be enhanced to support randomizing over sets of elementary functions and their compositions. Additional step is to make this process adaptive, i.e. only the functions that the learner already knows about, will be considered. Information about the prior knowledge of the learner has to be extracted from the learner model."

In other words our task is to extend it so that the possible values for the variables can be specified as intervals and that in other cases a variable can be replaced by a complete formula entirely generated by randomized composition of elementary functions from a given list which includes basic arithmetic and trigonometric functions. For the intervals we need to allow the content author to specify the extrema of the interval, whether each of them is closed or open, and whether the interval is discrete or continuous. For the formula generation we need to to ensure that the focus concept (the concept to be learned in the current session) is always used in the formula, and query the User Model in ActiveMath to avoid including any functions that the user does not know yet (apart from the focus concept). Further, we need to ensure that the term we generate is valid, that is, for example, we do not generate a fraction with a denominator that evaluates to zero.

3 Selection from Interval

The content takes the form of a new XML tag called 'interval', with the different combinations (open/closed, discrete/continuous) specified as attributes. We specified the changes in ActiveMath's DTD files. For instance, the discrete closed interval from one to ten [1, 10] is specified as follows:

```
<interval discrete="yes" left_open="no" right_open="no">
     <OMOBJ><OMI>1</OMI></OMOBJ>
     <OMOBJ><OMI>10</OMI></OMOBJ>
</interval>
```

Below we describe in turn how we handled the cases for discrete and continuous intervals in their four variants (open-open, open-closed, closed-open, closed-closed). The task of creating such intervals starting with one of them seems

deceptively simple. We describe in-depth how the situation is handled in the continuous case which turned out to be more involved than expected.

3.1 Discrete Intervals

For discrete intervals we use the integer random number generators included in the Java library and perform simple operations to get the four cases:

- $[a, b)$: this is provided directly in class Random of the Java libraries under java.util.Random.nextInt(int n).
- $(a, b]$: in this case we get a number in $[a, b)$ from the function specified above, and add 1 to the result to shift it.
- (a, b): now we get a number in $[a, b-1)$, and add 1 to it.
- $[a, b]$: here we get a number in $[a, b+1)$, which is always in $[a, b]$.

In practice a line for getting a number in a closed-open interval looks like this:

```
if (!has_open_left && has_open_right) {
    number = (int)java.lang.Math.floor(interval_start +
      (java.lang.Math.random() * (interval_end-interval_start))
    );
}
```

3.2 Continuous Intervals

The situation with continuous intervals is more complicated: computers cannot handle real numbers, so we approximate this case by using the corresponding concept in ActiveMath, double precision floating point numbers following the IEEE-754 standard. Normal random number generators, including the one provided by the Java platform, provide only numbers in the interval $[0, 1)$, in this case computed by modifying a 48-bit seed using the linear congruential method described by Donald Knuth in "The Art of Computer Programming", Volume 2: Seminumerical Algorithms, Section 3.2.1 pp. 10-26. Therefore, we can easily obtain a floating point number n in an interval $[a, b)$ from the number r we get from the Java randomizer as follows:

$$
\begin{aligned}
& r \in [0, 1) \\
& n = a + r(b - a) \\
& n \in [a, b)
\end{aligned}
$$

How to get the other cases?

For the case $(a, b]$, it suffices to flip the interval. First we obtain a number r_0 from Java, then take its symmetric through the center of the interval $[0, 1]$, which does not change the probability distribution, and then we compute n from it as above:

$$\begin{aligned} & r_0 \in [0, 1) \\ & r = 1 - r_0 \Rightarrow r \in (0, 1] \\ & n = a + r(b - a) \\ & n \in (a, b] \end{aligned}$$

For the cases $[0, 1]$ and $(0, 1)$ there is no trivial solution. We can take advantage of the limited precision of floating point numbers to compute an epsilon that works the same way as the unit in the discrete interval case (such a number is sometimes called a "machine epsilon" or a "ULP": "Unit in the Last Place"). Once we find it, we have:

$$\begin{aligned} & r \in [0, 1) \\ & n = a + r(b - a + \epsilon) \\ & n \in [a, b] \end{aligned}$$

$$\begin{aligned} & r \in [0, 1) \\ & n = a + \epsilon + r(b - a - \epsilon) \\ & n \in (a, b) \end{aligned}$$

How can we compute the epsilon? We want it to be the smallest distinguishable difference for the floating point numbers we are using. This number depends on the magnitude (the exponent part) of those numbers, so we have to compute it for the interval extremum to which it will be added or subtracted.

First we construct a double precision floating point number with the same exponent as the reference number, and the unit mantissa. Double precision floating point numbers following the IEEE-754 standard are encoded in 64 bits, with one bit for the sign, eleven bits for the exponent (some values of which are used for special values such as Not-a-Number and Infinity), and fifty-two bits for the mantissa. As the mantissa bits encode only the fractional part[1] (the 1 before the decimal point is implicit), the representation for the unit is actually 0x0000000000000000 so we don't have to set it explicitly. As for the sign bit, we also leave it out thus making the epsilon positive as needed in our formulas.

Once we have the unit mantissa with the same exponent as the reference number, we need to shift the number to the last bit of the reference mantissa, to reach its limit of precision. Since the mantissa is 52 bytes long, we need to decrease the exponent by 52 (0x34 in hexadecimal).

[1] We assume here that the reference number we receive is normalized

Our final implementation includes the following function that computes an optimal epsilon (ULP) for a given number[2]:

```
static double computeEpsilonFor(double n)
{
    long bits = Double.doubleToLongBits(n);
    long exponentBits = bits & 0x7ff0000000000000L;
    if (exponentBits > 0x0340000000000000L) {
        bits = (exponentBits - 0x0340000000000000L);
    }
    else if (exponentBits > 0x0000000000000000L) {
        bits = (0x01L << (0x34 - (exponentBits >> 0x34)) );
    }
    else {
      // In this case n is a subnormal number or zero.
      bits = 0x01L;
    }
    return Double.longBitsToDouble(bits);
}
```

Because of underflow, we have to produce a subnormal number (exponent 0) as result for values of n smaller than 2^{-971} in magnitude. In those numbers, there is no implicit leading digit 1, so we put the unit bit in the mantissa.

4 Random Function Composition

For the function composition we introduced another tag to the DTDs for the OMDOC format: "function_composition", which can be a parent tag for OMVs which will act as roots of the term rewriting tree. Below is an example of the extended format:

```
<parameter name="constant">
    <OMOBJ><OMV name="cool_function"/></OMOBJ>
    <function_composition>
        <OMOBJ><OMV name="x"/></OMOBJ>
        <OMOBJ><OMV name="y"/></OMOBJ>
    </function_composition>
</parameter>
```

The OpenMath objects inside the `function_composition` tag define the initial values of the variables used in the composed expression. We can refer to other randomized constants by using OpenMath variables as shown here, which are first replaced by their randomized values and then used for constructing the final expression. It is thus possible to cascade function compositions.

[2]We have noticed that an equivalent function is available in Java 1.5 as Math.ulp(), but we did our developent on Java 1.4.2.

4.1 Primitive Function Set

The functions below are the set of basic arithmetic and trigonometric primitives stated in the the task description:

- $x + y$, $x - y$
- xy, $\frac{x}{y}$
- e^x, $\ln x$
- $\sin x$, $\cos x$, $\tan x$, $\arcsin x$, $\arccos x$, $\arctan x$

Some functions take two parameters meaning that it is advisable to have two variables as in the example above. Otherwise the randomization will be over a single variable which is probably not desirable as it is very likely to have randomization resulting in $\frac{sinx}{sinx}$ for example.

Crucial in the term rewriting algorithms is the function *substituteVariables* (provided by the Exercise System in ActiveMath) that recives the substitution table as a *HashMap* and applies them to a given OpenMath *Element*. We wrap it in another function that takes care of locating the OpenMath objects inside an arbitrary XML tree.

It is applied with the α-renaming table (*alphaRenamingTable*) to avoid capture of variables, and also with the expression expansion table (*expansionTable*) that builds the randomized expression.

We use 0-based de Bruijn indexes for the variable names so that our algorithm scales to an arbitrary number of variables.

4.2 CAS Validation

Randomly generated expressions must be validated against a running CAS to ensure that, for instance, the denominator in a fraction is not zero valued. This is done through the infrastructure for communication with Computer Algebra Systems provided by ActiveMath. Three situations can arise:

- the denominator is not zero valued: no action taken
- the denominator is equal to zero but the nominator is not: reverse the fraction
- both the denominator and the nominator evaluate to zero: convert the division into multiplication

In this way we make sure that there is only one execution of the term rewriting algorithm per function generation request because every result will be a valid mathematical expression so we do not need to discard it and start again.

This checking and correction is clearly encapsulated in the function *checkExpressionConsistency()*, so it can be easily extended.

4.3 User Adaptation

In terms of user adaptation we provide three flexible mechanisms that ensure that users will get the correct exercise depending on the topic they are currently studying (focus function), their overall competency (difficulty, depth of the generated expression's tree) and the topics that the user has already covered (function candidate list filtering).

4.3.1 Filtering of the Function Candidate List

We can dynamically remove functions from the list of basic function specified above based on the knowledge of the student. To achieve this we query ActiveMath's User Model component to get the mastery value for a given concept (a function in this case), and include it in the generated formula only if the mastery of the user for it is greater than fifty percent:

```
private boolean userKnowsConcept(String conceptId){
    if (adaptKnowledge) {
        return (localUserModel.user.getMastery(conceptId) >= 50);
    }
    else {
        return true;
    }
}
```

Now, for instance, we can check if the user has enough mastery of the basic trigonometric function consinus and include it in the list:

```
if ("mbase://openmath-cds/transc1/cos".equals(conceptId)
    || userKnowsConcept("mbase://openmath-cds/transc1/cos")) {
    expression = oma(oms("transc1", "cos"), omv("0"));
    functionList.add(expression);
    if ("mbase://openmath-cds/transc1/cos".equals(conceptId)) {
        conceptIdExpression = expression;
    }
}
```

Here the list of functions *functionList* is an (*ArrayList*), *conceptId* is a string with the identifier of the concept we are checking for and *expression* is an OpenMath XML *Element*.

4.3.2 Focus Function

The focus function is a primitive function that is guaranteed to be used in the composition. That functionality has high pedagogical value as the content creator of exercises that are supposed to strengthen a student's ability to differentiate trigonometric functions, for example, would indicate that he would like

one of the primitive trigonometric functions from the list specified above should appear.

The positioning of the focus function in the term rewriting tree also randomized. That is, the focus function is guaranteed to be used but it can be applied at any iteration of the algorithm.

4.3.3 Depth Selection

The selection of the iteration depth depends on the level of difficulty to which we want to subject the student, specified in the exercise document.

There are five levels of difficulty that naturally correspond to expression tree depths from one to five. Below is a table containing an example term rewriting sequence up to a depth of five.

Metadata value	Expression tree depth	Example
very_easy	depth = 1	$\sin x$
easy	depth = 2	$e^{\sin x}$
medium	depth = 3	$\sqrt[3]{e^{\sin x}}$
difficult	depth = 4	$\frac{\tan x}{\sqrt[3]{e^{\sin x}}}$
very_difficult	depth = 5	$\ln\left(\frac{\tan x}{\sqrt[3]{e^{\sin x}}}\right)$

The term rewriting loop is iterated *depth* times to get the final expression.

4.4 Architecture Overview

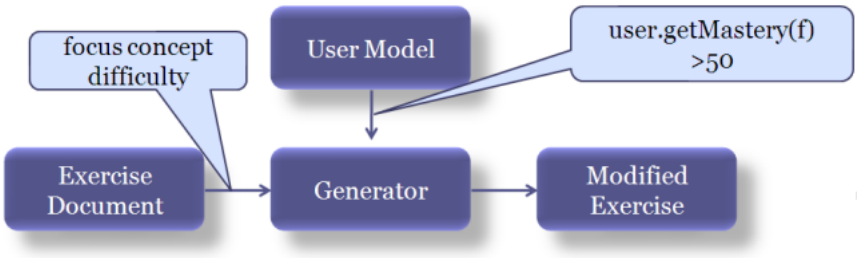

Figure 2: Overview of the way the Randomizer interacts with other components in ActiveMath.

5 Questions raised during our presentation

5.1 Necessity of computing the epsilon for continuous intervals

As we presented our algorithm for computing random numbers in (pseudo-) continuous intervals, it was proposed that we could handle the closed-close $[0, 1]$ case from $[0, 1)$ much more easily by "adding one" instead. This turned out to mean choosing randomly between the upper extremum and the closed-open interval, and then in the second case picking a random number from it as before.

Our objection was that such procedure introduced an enormous bias in the distribution, but there was no time then to elaborate on it as other presentations had to be done in a limited time. We explore here that idea in more detail.

By first choosing between the upper extremum and the closed-open interval, the probability of getting the upper extremum is $\frac{1}{2}$, which is typically much greater than for any other number in the interval (for a truly continuous interval, the probability is actually $\frac{1}{\infty}$).

The proposed way to correct the obvious biasing was to weight the probability for the upper extremum. The remaining question is: how much?

The objective is to make the upper extremum be exactly as probable as any other point in the interval. For floating point numbers:

$$P[r = s, s \in [a, b]] = \frac{\epsilon_{b-a}}{b - a}$$

where ϵ_n is the computed epsilon for a given number n. For $[0, 1)$, $a = 0$ and $b = 1 - \epsilon_{b-a}$, thus:

$$P[r = s, s \in [a, b]] = \frac{\epsilon_{(1-\epsilon_{b-a}-a)}}{1 - \epsilon_{b-a} - a}$$

Therefore we need to compute the epsilon anyway, so it is not easier than our proposed algorithm.

5.2 Justification for the Difficulty

Difficulty can not be defined in absolute terms, so we used a simple mapping from the difficulty levels in ActiveMath to the tree depth of the produced expressions. We assume that, given an expression, putting it as argument to a function makes the new expression more difficult, but even such a simple assumption is not always true: for any finite expression e, $0/e$ is trivially 0, so it could be the case that an expression was actually made easier by iterating the composition algorithm.

Solving this in general is out of the scope of this work, so we provide a simple solution that works well in many cases, and leave the implementation open for further work in this area.

6 Conclusion

In this report we presented the extensions we implemented for the Randomizer class in ActiveMath.

We implemented random selection from intervals (open, closed, discrete, continuous), by meticulously taking care to provide the most accurate solution to make available all interval types without biasing the distributions of the random values more than the strictly necessary.

As a second subtask we implemented the composition of arbitrarily complex expressions from a list of basic arithmetic and trigonometric functions. Our implementation takes into account the user model by adapting the generated functions in three ways: iteration depth (depending on the difficulty), focus function (depending on the target concept) and filtering of the function candidate list (depending on the user's past knowledge).

In our version of the Randomizer class, 80% of the lines are either new or modified. Furthermore we have taken special care to write high-quality, readable and extensible code.

7 Authors' Contributions

7.1 Final Project

The two authors spent close to ten hours working together to provide the basic functionality for the randomizer. During this time they implemented the random interval selection and the basic expression generation.

Alberto González Palomo further spent over eight hours on providing user adaptation and refining the implementation. He also provided invaluable insights into the architecture of ActiveMath without which the implementation and testing time would have been at least double.

Minko Dudev spent close to four hours (two of which together with Alberto González Palomo) preparing the class presentation and over six hours preparing this report (two of which together with Alberto González Palomo).

7.2 OMDoc Encoding

The two authors spent a countless number of hours listening to the recording of the lecture and transcribing it.

Minko Dudev spent over five hours refining the language of the transcript, cropping images and organizing it in a readable way.

Alberto González Palomo spent five hours converting the initial QMath encoding to OMDOC, incorporating images, and adding final touches to the formatting.

Both authors spent over an hour fixing the image paths when there was a change in the indexing system of the ActiveMath server on which the students' lecture transcripts had to be displayed.