Publication V

Otto Seppälä. Modelling Student Behavior in Algorithm Simulation Exercises with Code Mutation. In *Proceedings of the 6th Annual Finnish / Baltic Sea Conference on Computer Science Education.*, pages 109–114, November 2006.

© 2006 ACM.
Reprinted with permission.

Modelling Student Behavior in Algorithm Simulation Exercises with Code Mutation

Otto Seppälä
Helsinki University of Technology
PL5400, 02015 TKK, Finland
oseppala@cs.hut.fi

ABSTRACT

Visual algorithm simulation exercises test student knowledge of different algorithms by making them trace the steps of how a given algorithm would have manipulated a set of input data. When assessing such exercises the main difference between a human assessor and an automated assessment procedure is the human ability to adapt to the possible errors made by the student. A human assessor can continue past the point where the model solution and the student solution deviate and make a hypothesis on the source of the error based on the student's answer. Our goal is to bring some of that ability to automated assessment. We anticipate that providing better feedback on student errors might help reduce persistent misconceptions.

The method described tries to automatically recreate erroneous student behavior by introducing a set of code mutations on the original algorithm code. The available mutations correspond to different careless errors and misconceptions held by the student.

The results show that such automatically generated ”misconceived” algorithms can explain much of the student behavior found in erroneous solutions to the exercise. Non-systematic mutations can also be used to simulate slips which greatly reduces the number of erroneous solutions without explanations.

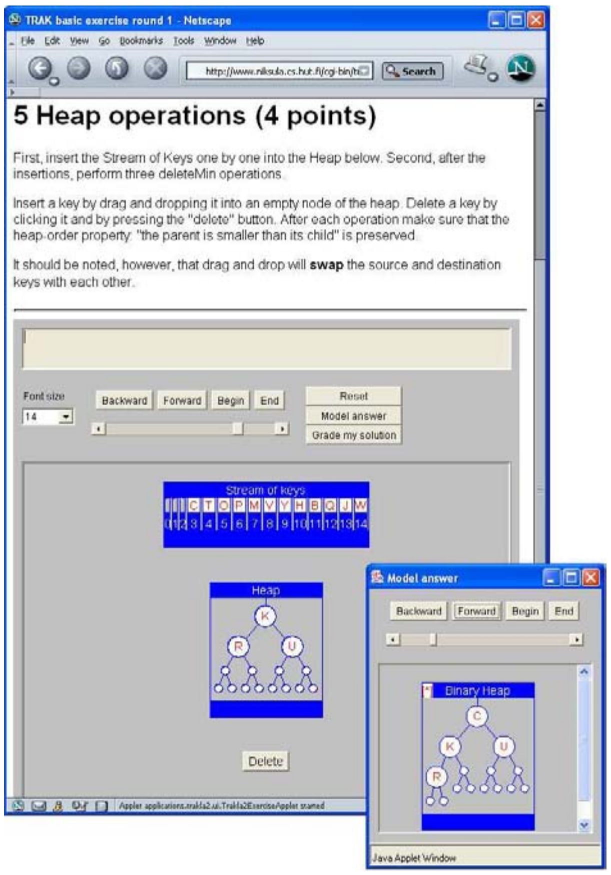
1. INTRODUCTION

On the Data Structures and Algorithms courses in the Helsinki University of Technology we use the TRAKLA2 system[7] to assess how well students know how different algorithms taught on our course should operate. Rather than requiring the students to implement these algorithms, the system tests their knowledge using *visual algorithm simulation exercises*. These exercises are then automatically graded and form a part of the course grade. While automatic assessment saves us hours and hours of assessment time it also gives the students the possibility of getting immediate feedback on their solutions during day and night.

This far the feedback has typically consisted only of the number of correct steps and a model solution. As the student solution is also still available, an interested student has been given a possibility to review the answer against the solution and figure out what went wrong. For some time now we have been researching on how to improve the quality of this feedback and essentially it all boils down to being able to interpret the error made by the student.

Our previous paper[8] on the subject studied the possibility of simulating the errors by manually implementing algorithm variants that correspond to different misconceptions. The approach described in this paper extends on this work with a way to automatically generate some of the algorithm variants as well as a method to handle careless errors.

©Copyright held by the author(s).
ISBN: 978-1-59593-970-8



5 Heap operations (4 points)

First, insert the Stream of Keys one by one into the Heap below. Second, after the insertions, perform three deleteMin operations.

Insert a key by drag and dropping it into an empty node of the heap. Delete a key by clicking it and by pressing the "delete" button. After each operation make sure that the heap-order property: "the parent is smaller than its child" is preserved.

It should be noted, however, that drag and drop will **swap** the source and destination keys with each other.

Figure 1: TRAKLA2 applet page and the model solution window. In this exercise the heap operations can be simulated by moving the keys in the data structures using a mouse.

2. VISUAL ALGORITHM SIMULATION EXERCISE ASSESSMENT

A visual algorithm simulation exercise typically includes a number of data structures in randomly generated initial states and an exercise description which explicitly tells which algorithm to follow. Depending on the exercise being solved, the student can use use the mouse to perform different drag and drop operations which again swap keys, change edges in dynamic structures etc. Each operation that changes a data structure is recorded to the *student solution sequence*. This sequence is later on graded by checking it against a *model solution sequence* generated by an existing implementation of the algorithm.

The normal method of assessing these exercises is to first select a state from the model solution sequence and then browse through the student sequence until a matching state is found. The comparison process then advances to the next state in the model sequence and searches for a matching state in the student sequence from the states following the last match found. The process is repeated until we run out of states in either sequence. The points from the exercise are proportional to the amount of model sequence states matched by states in the student sequence.

Typically the model sequence is shorter than the student sequence. In some cases the student has freedom to choose in which order to perform some minor operations, the order of which is not explicitly stated in the algorithm. A swap operation for example allows two different orderings. To avoid useless iteration we can allow all such orderings by only requiring the states where all minor operations have already been performed.

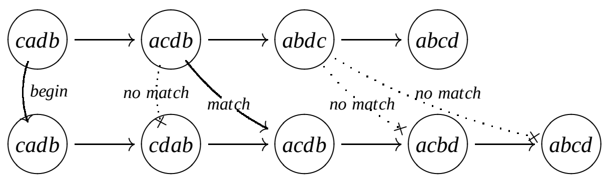


Figure 2: Comparison between two sequences. (Selection-sort of an array) The top sequence is the model, the bottom the student sequence. Two matches were found.

One limitation of the current assessment procedure for the visual algorithm simulation exercises is that while it can count the number of correct steps is a sequence, it offers no information on what was the cause behind the incorrect step in the algorithm simulation. It also cannot find out if any steps following such erroneous state could have been correct relative to the incorrect state created by an earlier mistake made by the student.

When similar algorithm tracing exercises are solved in a paper exam, a human examiner can often tell if the errors are just random slips or more systematic errors possibly resulting from misconceptions. Telling the two apart is important for giving students proper feedback. A human examiner can also continue the assessment past a small careless error. One aim of our research project is to try and bring some of this ability also to automated assessment.

3. INTERPRETATION OF ERRORS

After the student has submitted her solution she can view the model sequence for that exercise instance and compare where and how the answer sequenced deviate. As explained, the problem with interpreting the model solution sequence is, that while locating the differing state in the sequences is easy and straightforward, finding out the reason behind this difference is not. For that, a student (or the examiner) should actually repeat solving the exercise. This is because the errors that were made, were made in a specific *context* created by the previous steps of the algorithm – to interpret them, that same context is needed. Analogously, if you begin reading a novel from the middle, understanding the next action in the plot is often somewhat impossible.

The same is essentially true when trying to reinterpret a recorded student solution or even an answer written on an exam paper. As the recorded data structures do not include many of the auxiliary variables used nor the position in the algorithm code, the assessor must also start the tracing of the algorithm from the beginning to understand the error made by the student.

3.1 Automatic Context Creation in the Presence of Misconceptions

Artificially recreating the interpretation context seems a fairly straightforward task. An implementation of the algorithm can be run and its data structures compared until we find the last state that matches the recorded sequence. At that time the implementation holds the “full” algorithm state with also the variables that were not originally recorded in the student sequence. Such variables include loop variables etc. which are not shown or input to the system by the student. The reconstructed context can then be used to better evaluate the error the student made.

As the context is directly dependent on the algorithm being traced, it is also affected by any misconceptions the student holds about that algorithm. This again requires that contexts are created for misconceived algorithms as well.

In [8] we tested if student misconceptions about algorithms could be modeled by manually implementing *variants* of the algorithm being studied. The misconceptions could then be identified by using the same assessment method that is used to assess the student solutions. Essentially, the user solutions were tested against each of the variants to see if any of the sequences created by the variants better matched what happened in the student sequence. The results from the study showed that the approach is quite usable for pointing out popular misconceptions. An exhaustive search for less popular variants and implementing them however seems not feasible or is at least very laborious to do manually.

While the approach described is able to find exact matches to any of these variants, it cannot recognize or handle *careless errors*, which are often only small alterations in the form of skips, off-by-one errors, misreading alphabetic order and such. It would however be possible to model skips by creating a separate variant for each and every skip or combination of skips. It is quite clear that manual implementation can not be done given the number of possible combinations.

Even if we do not consider the careless errors, manually implementing each prospective candidate also takes a lot of work. The tools used for tackling both of these problems to a certain degree and automatically creating algorithm variants are introduced in the next section. In section 5 we define two assumptions which define the area of applicability. Section 6 describes the actual implementation used. Results from an experiment on recorded answer sequences are given in section 7. Section 8 discusses future work.

4. RELATED WORK

An approach to recognizing misconceptions by making alterations on a model of a skill was used by the BUGGY[2] and DEBUGGY[4] systems. The systems tried to infer misconceptions held by pupils learning the basics of in-place subtraction. The system had a model of subtraction that was divided into subskills that could be replaced by their incorrect counterparts. If the subtraction problem was carefully selected, a matching result generated by the altered model could point out students who for example had a specific misconception of how to borrow.

Another influencial system to be mentioned is the LISP Tutor by Anderson et al.[1]. LISP Tutor had a model that would perform the task the student was expected to perform. While the student solves a problem, correct and incorrect steps are immediately recognized. In case of errors the student is given instruction that tries to steer the student back to a correct path. Their original solution to interpreting what the steps made by the student actually meant was to provide the student with a disambiguation menu, which provided the model with information if could not infer from the students actions.

5. CODE MUTATION

The automatic variant creation described in this paper is done by introducing changes in the code of the original implementation[1]. Such controlled changes are often used in applications such as *mutation testing, fault injection* and *genetic programming*. The methods we will use for creating the algorithm variants have been originally introduced and extensively studied for use in mutation testing. As a result we will also use some of the vocabulary in that area.

5.1 Background

Mutation testing is an idea proposed by DeMillo et al,[5]. It aims at evaluating and improving test data by introducing changes, *mutations*, to the program code which the test cases should then be able to find. A test case capable of killing an introduced bug is potentially able to find other bugs in the bug's neighborhood.

5.2 Mutant Creation

The changes to the program code are made with *mutation operators*. A mutation operator is a change which typically replaces a small portion of the program code with different code. A classic example would be to interchange any of the arithmetic operators $+ - / *$ with another arithmetic operator at one point in the code. A program changed with at least one mutation is called a *mutant*.

An instrumented code that contains all the possible mutations controllable at runtime by the testing system is called a *metamutant*[9]. The main advantage of this approach is that the code is only compiled once. The downside is that the the code executes slower. In the prototype described in this paper the metamutant approach is used. We will use the name *mutation point* when referring to portion of code in a metamutant which is changeable with a mutation operator.

6. ASSUMPTIONS

The misconception modeling approach makes two assumptions of the students and their knowledge which are essential for this approach to work. The assumptions both define the area of application and are also used when pruning the search tree in the mutation search for the best candidates.

6.1 Systematicity Assumption

It is quite safe to assume that university students know at least one thing about the algorithms taught on the course – that algorithms are executed in some systematic way. Therefore even in the presence of a misconception it should hold that the whatever algorithm the student is trying to follow, it would still be systematic. We just do not know which algorithm it is. Although there exist students that might not share this understanding, we do not have to consider them as their problems are too profound to be tackled with algorithm simulation exercises.

Exceptions to this systematicity rule are the unintentional careless errors made. After the careless error the sequence should continue with normal steps created by the algorithm, be it the correct or a misconcepted one.

It is also important to point out that as the TRAKLA2 exercises often require the student to repeat the algorithm on a set of data instead of a single key etc., the systematicity of possible misconceptions shows up quite well even for algorithms that would normally have a relatively small number of steps.

[1]and possibly also the code of any manually implemented variants

6.2 Mutation Distance Assumption

The second assumption is that as many of the misconcepted algorithms are derived from the original algorithm, the implementation of the misconcepted algorithm is not far from the implementation of the correct algorithm.

In a sense this is related to the *competent programmer hypothesis*[3] which is one of the cornerstones of mutation testing. The hypothesis is that competent programmers should be able to create programs that only differ from a perfect program by a given distance. The hypothesis is essential for the mutated program to efficiently model the real-world faults the test cases should be able to catch.

We know however from data collected from students that the mutation distance assumption does not always hold. Manual search through the answer sequences has shown that misconcepted algorithm variants exist that have notably more complex implementations than the original algorithm. The original algorithm might for example require no external storage whereas the student algorithm might require a dynamic memory structure such as a stack to be implemented.

This is not surprising as the synoptical view onto the data structure easily misleads students uncustomed to working with data structures by hiding the inherent complexity behind a seemingly simple approach to a problem. A good example is performing a sort, which can be performed by anyone, with or without any education in sorting algorithms.

This finding only implies that the mutation methods must be backed up with some manual implementation of the more distant algorithm variants. Mutation can then be used to find the subvariants in their vicinity.

6.3 Implications of the Assumptions

In this research we concentrate on the misconcepted algorithms that fill both of these assumptions. It is clear that if the systematicity assumption does not hold, the sequences generated are uninteresting as randomly trying out the algorithm typically is not a sign of a misconception about something learned but more of a wild guess. There is no point in generating guiding feedback from guesses.

For the second assumption to hold, the first one must already be true. As previously pointed out, the second assumption can be violated but the misconcepted algorithms could still be systematic.

We have demonstrated in [8] that it is possible to sieve out likely candidates for misconcepted algorithms and then implement these by hand. In this paper we however are interested in algorithms that can be derived from the original with more minute changes.

The next section explains the mutant creation and mutation search procedure in more detail.

7. IMPLEMENTATION

For the prototype we have chosen to use the metamutant approach. The code for the metamutant cannot currently be created automatically. The algorithms are therefore prepared by hand, inserting the *mutation points* where appropriate. Knowledge on the algorithm operation is of assistance when choosing the mutation points as all mutations do not lead to sensible code, but only create useless execution.

The series of mutation choices made at each mutation point when executing the metamutant is called a *mutation sequence*. This is not to be mistaken with the student sequence and the model sequence which store the states of the data structures. The process of finding the mutation sequence which best explains the recorded student sequence is called a *mutation search*.

7.1 Mutation Operators

A *mutation operator* is a description of a syntactically correct change to an existing program that will change the semantics of that program, resulting in a new program, called a *mutant*

Typically mutation operators are designed to make single-point changes to the source code of the program. The changes can be made prior to compilation, or at runtime if the code has been instrumented to host a number of mutations that can be switched on and off on demand.

Our approach is different from the normal use of mutation operators, as we want to change the state of the mutation run-time. This is required to model the careless errors that break the systematicity of the algorithm.

For this prototype implementation we have chosen to use only a small number of mutation operators, which are decribed below.

- *Zero Operator* is a restricted form of the more general integer offset operator used by many mutation testing systems. It normally evaluates to zero, but can also evaluate to one, making it useable for modeling off-by-one errors, skips etc. depending on the place where it is applied. Respectively there also exists a ”one”-operator.
- *Comparison Operators* change the result of the corresponding boolean comparison to its negation. Although we normally have 6 different comparison operators to choose from, the mutation operator works perfectly well working only with the original form and its negation. This limits unwanted forking of the mutation search tree.

  Later on, when we want to investigate the final candidates, we can infer from the variable values, which of the comparison operators would have returned true or false when the comparison was made.
- *Arithmetic Operators* change additions to substractions and vice versa. Divisions can be rounding instead of truncating. Using a rounding division is a fairly common novice mistake.
- *Skip Operator* essentially is a boolean value used to control a conditional clause which can be used to skip a portion of the code.

The points in the code where steps are added to the model solution sequence are marked with code similar to the mutation points. These code locations, *Animation points*, are important for pruning (explained in 7.4), as they are the only possible locations where new steps are added in the solution sequence and are thus the only places where the score of the comparison between the student sequence and the generated one could increase.

7.2 Mutation Recorder

The mutation recorder is used to record the mutations made during the execution sequence. Each time a mutation point is passed, it creates a new link, *mutation step*, in the mutation sequence. Such links are also made when the mutation point operates like the original non-mutated code. The specific mutation is chosen by the mutation recorder based on the previous mutations taken that followed this same path.

The information on the child nodes expanded is stored in the mutation step objects as well as which child of the father node this node is. All the mutation sequences together form a father-linked tree where each mutation operation links to the previous one. This approach not only reduces memory consumption, but also allows a mutated execution sequence to be referenced only using the link created by the last mutation operation.

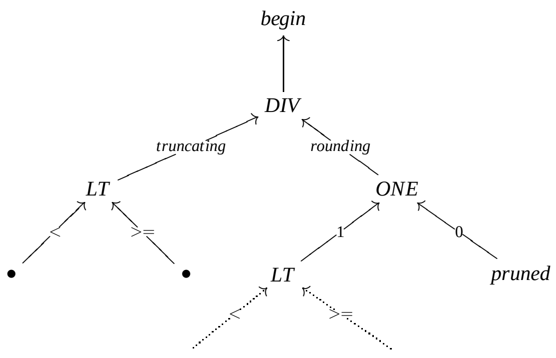


Figure 3: A tree containing mutation sequences

The recorder is also used to replay the changes when a variation of a previous sequence is wanted. This functionality is required when we want to derive a new sequence from a previous one. In this case a part of a mutation sequence is rewound into a stack inside the mutation recorder. When replaying, the mutation points act the same way as when the original sequence was recorded. When the steps in the stack are finished, the recorder goes back to its normal operation.

7.3 Search Algorithm

The search algorithm attempts to generate an algorithm variant through mutation, which best explains the student's answer sequence. This is a simple depth first search with backtracking. Pruning the search tree nodes is used to make the search both feasible and also faster. The search algorithm finds first an initial solution, be it full or partial[2]. Typically the first try uses the original, unmodified algorithm. It then backtracks until a link is found with children that have not yet been expanded. Path from the start to this node is rewound to the mutation recorder for playback as explained in the previous paragraph.

The solving process is repeated with the steps stored in the recorder. When they end, the next node always is a node with unexpanded children. The recorder then selects a new child and the solving process continues expanding new nodes until the procedure is stopped the next time. The stopping conditions are given in the next section. The search algorithm will go through all possible mutation sequences up to the normal ending point of the algorithm or a point where the search is pruned or ended early by an exception.

7.4 Pruning the Mutation Search Tree

If the mutation search is performed without limiting the search in any way, the search would never end. Not only is the amount of branching between the beginning and the end very high even for a normal case, but there also exist mutated execution sequences that result in non-terminating execution. The mutations can also

[2]Partial solutions do not explain all student sequence steps, because they are pruned

lead to exceptions or early termination which are considered here as forms of "self-pruning". These are however considered to be a positive side-effect as they are quite effective in reducing the search space.

A partial solution to the problem of high branching and endless execution exists in the form of two pruning operators that have proven to be quite effective:

- *Limiting inconsistency in mutations made in a single execution*

  This pruning rule is a straight consequence of the systematicity assumption. If a mutation operator constantly changes its function, the algorithm is not anymore systematic. A limited number of deviations are allowed which model the careless errors done by a student.

- *Limiting the number of changes to the data structures which do not lead to a higher score*

  As with the traditional checking algorithm used by TRAKLA2, we have to take in account that the model algorithm only contains the major states between which the minor states can be scrambled. Scrambled or not, a limited number of changes to the data structures checked should lead to the next major state. This again should raise the score gained from the exercise.

  This pruning rule stops the search in a branch that has too many intermediate states following a "score state" that produce no change to the score. It is important to note that while the number of such intermediate states is specific to each algorithm, the value typically is quite low and controllable by the exercise designer.

7.4.1 Exceptions

It is not uncommon for the mutated algorithm to crash with an exception. Mutations violate the assertions the original algorithm follows. As the mutated algorithm implementation is seen as a model of the student's simulation, the exception thrown in the program code is essentially also an operation that could not be simulated. Sequences leading to exceptions can therefore be pruned.

7.4.2 Normal Termination

If the mutated execution is not pruned or does not end with an exception the result can either be a successful interpretation of the simulation sequence or an early termination, in which case we can prune the ended sequence if there was earlier an other sequence with a higher score. All terminating sequences are therefore evaluated and the best candidates chosen for closer inspection.

7.4.3 Other Cases

There exist cases that are not pruned by either of the proposed pruning operators, do not normally terminate, and do not throw exceptions. Such cases fall into three categories of non-terminating executions.

1. non-terminating execution path with animation points
2. non-terminating execution path with mutation points
3. non-terminating execution path with no mutation points

The first two could be pruned using similar pruning filters which are checked when a mutation point is reached. Even then, deciding the level of when to prune is not easy. For example, if a sorting algorithm is executed on an already ordered data structure, there would still have to be at least $O(n)$ comparisons made just to verify that the array is sorted. It is likely that there would be at least this amount of animation points passed where there are no changes to the data structure. Correspondingly there should be an even higher number of mutation points passed on the way, where pruning should not be done.

The last category, non-terminating execution path with no mutation points, cannot be pruned using any pruning techniques introduced this far, as pruning is only made in the annotated parts of the code.

The prototype therefore currently requires the programmer to recognize such points and write assertions that throw exceptions or force the code to eventually end in such situations.

8. RESULTS

To evaluate the approach used, a set of real recorded solution sequences from our data structures and algorithms course was used. The effectiveness of the mutation-based method was compared with the method used in [8]. The exercise used in the study was binary search as hand-implemented variants of misconceptions on that exercise already existed. These variants modeled misconceptions about truncating division and movement of the left and right pointers in the algorithm.

The hand-implemented variants were able to explain 40% of the sequences that were nor fully correct or completely empty (nothing done). For the mutation method the amount of solutions with at least one explanation was 65%. The improvement was mostly from the inconsistent mutations which were not possible to model in the manually implemented variants. The consistent mutations are in line with the hand-implemented variants although for some solutions the mutation method was able to find a simpler explanation using a single inconsistent mutation in place of two consistent ones.

It is important to mention though, that the binary search exercise is exceptional in the sense that all the hand-implemented algorithms were found using consistent mutations. For most algorithms it is likely that we still need to implement many of the variants by hand. The mutation approach can then be used to find minor deviations from these algorithms and to recognize slips.

8.1 Effect of Pruning

Both the two pruning strategies are of importance if we want to provide the student with immediate feedback. If no pruning is used the combinatorial explosion ensures that finding a solution is not feasible for a reasonable-sized exercise instance. Allowing one slip and one additional step made by the solution algorithm cuts the mutation search time down to 3 seconds per solution. Allowing more errors in the sequence might multiply the time by ten for each new error allowed. The pruning values are dependent on the exercise instance. The effect on the values on the solutions found is a matter of another study.

8.2 Known Limitations

One known limitation of the approach is that in many cases some seemingly simple systematic changes done by the students violate the mutation distance assumption. A good example of this is was found in the binary tree in-order traversal exercise. The normal in-order traversal recursively traverses the left subtree, then visits the node itself, and then traverses the right subtree. In some student sequences the node is visited after the right subtree if the left tree is missing. It is possible that not having the left subtree contradicts with a simplified model for traversal: left-node-right. The right subtree is then used in place of the left.

Such a simple-looking change adds to the complexity of the traversal algorithm in a way that cannot easily be handled with mutation. One could argue that a mutation operator capable on reordering code lines would be able to generate the desired mutant, but even then the mutation had to be conditional, depending on the value of the reference to the left subtree.

It is therefore likely that the main variations of the algorithms must still be prepared manually, but combining the manual approach with the mutations is a promising tool for examination of the solutions.

9. CONCLUSIONS

When students solve algorithm simulation exercises, the solutions form a sequence of states. Previously we have shown that a considerable number of these sequences can be explained by corresponding algorithm variants that model some typical misconceptions. We have now described an automatic way of generating variants from existing algorithm implementations through code mutation. This is an advantage over the previous results, as allows for more easily creating a variety of different versions of the original algorithm. It also allows modeling of random carelessness errors with non-systematic mutations.

Initial results suggest that introducing the carelessness error to the modeling sometimes allows for simpler explanations of students' simulation sequences.

10. FUTURE WORK

The next logical steps are to enhance the analysis of the results and correspondingly also the feedback to the students. Secondly the quality of the system and the feedback should be evaluated with students. It should also be possible to create the metamutant automatically.

10.1 Enhanced Feedback

One of the most interesting parts of the project comes when the error is presented back to the student. The aim is to use a visualization of the pseudo-code of the algorithm to point out the place where the student execution diverged from the model solution. This information visualized is actually the types and places of the mutations made to the code.

Another possibility is to write textual feedback that matches pattern of mutations found by the prototype. The problem with this is the possible conditionality of the mutation – that the mutation happens only when a specific condition is true. Therefore in most cases it seems that showing the position in the pseudo-code along with the type of change made by the mutation operator seems as the best alternative for now.

10.2 Evaluation

As the results of the trial runs with real course data have been promising, the next logical step is to evaluate the quality of the results online. Although the design of the evaluation is open, one possible way could be to first use multiple choice questions to probe the knowledge and misconceptions held by the student. These questions would be presented immediately after the user has submitted a solution, but before showing the exercise results. After the initial questions have been answered, we can show the results and the automatically generated feedback, the quality of which the user can then evaluate. Also, as the student would not be aware of any slips before seeing the results, an additional question about them would be presented as well.

10.3 Heuristics for Breaking Ties

Heuristics for selecting the best candidate are required when there are multiple good mutation candidates that equally well match the sequence. Such decisions are often case-specific, e.g. whether two consistent mutations are better than one inconsistent, depends a lot on the exercise instance and the specific mutation operators.

10.4 Tools

Automatic creation of the metamutant requires a simple Java parser to be built. As there is no actual need to understand the semantics of the code, this should be possible to do with only a little more than simple pattern matching and replacement. The additional requirements are assuring that the code is syntactically correct and checking that the mutation point must also operate in the exact same way as the original code did.

The infinite execution problem currently requires attention from the exercise designer. The placement of the assertions that break the infinite loops should also be possible to do automatically. This requires recognizing iterations and recursions in the code that fall in the three categories decribed in 7.4.3, and placing assertions in the loops that at least calculate and limit the number of iterations made.

11. ACKNOWLEDGEMENTS

This work was supported by the Academy of Finland under grant number 210947.

12. REFERENCES

[1] J. R. Anderson, A. T. Corbett, K. R. Koedinger, and R. Pelletier. Cognitive Tutors: Lessons Learned. *The Journal of the Learning Sciences*, 4(2):167–207, 1995, Lawrence Erlbaum Associates, Inc.

[2] J. S. Brown and R. B. Burton. Diagnostic models for procedural bugs in mathematical skills. *Cognitive Science*, 2:155–192, 1978.

[3] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–233, New York, NY, USA, 1980. ACM Press.

[4] R. B. Burton. Debuggy: Diagnosis of errors in basic mathematical skills. In *Intelligent Tutoring Systems*. Academic Press, 1981.

[5] R. A. Demillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 11:34–41, 1978.

[6] A. Korhonen. *Visual Algorithm Simulation*. Doctoral dissertation (tech rep. no. tko-a40/03), Helsinki University of Technology, 2003.

[7] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, 3(2):267 – 288, 2004.

[8] O. Seppälä, L. Malmi, and A. Korhonen. Observations on Student Misconceptions - A Case Study of the Build-Heap algorithm. *Computer Science Education*. 16(3):241 – 255, September 2006, Routledge.

[9] R. H. Untch. Mutation-based software testing using program schemata. In *ACM-SE 30: Proceedings of the 30th annual Southeast regional conference*, pages 285–291, New York, NY, USA, 1992. ACM Press.