

Software Verification and Graph Similarity for Automated Evaluation of Students' Assignments

Milena Vujošević-Janičić · Mladen Nikolić ·
Dušan Tošić · Viktor Kuncak

Received: date / Accepted: date

Abstract In this paper we promote introducing software verification and control flow graph similarity measurement in automated evaluation of students' programs. We present a new grading framework that merges results obtained by combination of these two approaches with results obtained by automated testing, leading to improved quality and precision of automated grading. These two approaches are also useful in providing a comprehensible feedback that can help students to improve the quality of their programs. We also present our corresponding tools that are publicly available and open source. The tools are based on LLVM low-level intermediate code representation, so they could be applied to a number of programming languages. Experimental evaluation of the proposed grading framework is performed on a corpus of university students' programs written in programming language C. Results of the experiments show that automatically generated grades are highly correlated with manually determined grades suggesting that the presented tools can find real-world applications in studying and grading.

Keywords automated grading, software verification, graph similarity, computer supported education

This work was partially supported by the Serbian Ministry of Science grant 174021 and by Swiss National Science Foundation grant SCOPES IZ73Z0_127979/1.

Milena Vujošević-Janičić · Mladen Nikolić · Dušan Tošić
Faculty of Mathematics, University of Belgrade, Belgrade, Serbia
E-mail: milena@matf.bg.ac.rs
E-mail: nikolic@matf.bg.ac.rs
E-mail: dtosic@matf.bg.ac.rs

Viktor Kuncak, School of Computer and Communication Sciences, EPFL, Station 14, CH-1015 Lausanne, Switzerland
E-mail: viktor.kuncak@epfl.ch

1 Introduction

Automated evaluation of programs is beneficial for both teachers and students (Pears, Seidman, Malmi, Mannila, Adams, Bennedsen, Devlin, & Paterson, 2007). For teachers, automated evaluation is helpful in grading assignments and it leaves more time for other activities with students. For students, it provides immediate feedback which is very important in process of studying, especially in computer science where students take a challenge of making the computer follow their intentions (Nipkow, 2012). Immediate feedback is particularly helpful at first programming courses where students have frequent and deep misconceptions (Vujošević-Janičić & Tosić, 2008).

Benefits of automated evaluation of programs are even more significant in the context of online learning. A number of world's leading universities offer numerous online courses. The number of students taking such courses is measured in millions and quickly growing (Allen & Seaman, 2010). In online courses, the teaching process is carried out on the computer, the contact with teacher is already minimal and hence the fast and substantial automatic feedback is especially desirable. Therefore, automation of evaluation tasks in online learning is very important.

Most of the tools for automated evaluation of students' code are based on automated testing (Douce, Livingstone, & Orwell, 2005). Testing is used for checking functional correctness of student's solution, i.e., whether the student's program exhibits the desired behavior on selected inputs. Testing can also be used for detecting bugs. We consider bugs to be runtime errors and exclude errors that only compromise functional correctness (for example, in programming language C, some important bugs are buffer overflow, null pointer dereferencing and division by zero). Although there is a variety of software verification tools that could enhance automated bug finding in students' programs (by analyzing the code without executing it), these tools are usually too complex to use and cannot be easily adapted for educational purposes.

In addition to checking functional correctness, an evaluation tool may also analyze program efficiency and/or complexity by profiling. Relevant aspects of program quality are also its design and modularity (adequate decomposition of code to functions). These issues are addressed by checking similarity to a teacher provided solution. In order to check similarity, aspects that can be analyzed are: frequencies of keywords, number of lines of code, number of variables etc. Recently, a more sophisticated approach of grading students' programs by measuring the similarity of related graphs has been proposed (Wang, Su, Wang, & Ma, 2007; Naudé, Greyling, & Vogts, 2010). Recent surveys of related approaches are given elsewhere (Ala-Mutka, 2005; Ihantola, Ahoniemi, Karavirta, & Seppälä, 2010).

In this paper, we propose a new grading framework for automated evaluation of students' programs aiming primarily at introductory programming courses. The framework is based on merging information from three different evaluation methods: it merges results obtained by software verification (automated bug finding) and control flow graph (CFG) similarity measurement with results obtained by automated testing. The synergy between automated testing, verification, and similarity measurement improves the quality and precision of automated grading and overcoming the individual weaknesses of these approaches. Our experimental results show that our framework can lead to a grading model that highly correlates

to manual grading and therefore gives promises for real-world applicability in education.

We also briefly discuss tools for software verification (Vujošević-Janičić & Kunčak, 2012) and CFG similarity (Nikolić, 2013), that we use for assignment evaluation. These tools, based on novel methods, are publicly available and open source.¹ Both tools use the low-level intermediate code representation LLVM. Therefore, they could be applied to a number of programming languages and could be complemented with other existing LLVM based tools (e.g., tools for automated test generation). Also, the tools are enhanced with support for meaningful and comprehensible feedback to students, so they can be used both in the process of studying and in the process of grading assignments.

Overview of the paper. Necessary background information is given in Section 2. Motivating examples for the synergy of the three proposed approaches are given in Section 3. The grading setting and the corpus used for evaluation are described in Section 4. The role of the verification techniques in automated evaluation is discussed in Section 5 and the role of structural similarity measurement is discussed in Section 6. An experimental evaluation of the proposed framework for automated grading is presented in Section 7. Section 8 contains information about related work. Conclusions and outlines of possible directions of future work are given in Section 9.

2 Background

This section provides an overview of intermediate languages, the LLVM tool, software verification, the LAV tool, control flow graphs and graph similarity measurement.

Intermediate languages and LLVM. An intermediate language separates concepts and semantics of a high level programming language from low level issues relevant for a specific machine. Examples of intermediate languages include the ones used in LLVM and .NET framework. LLVM² is an open source, widely used, rich compiler framework, well suited for developing new mid-level language-independent analyses and optimizations of all sorts (Lattner & Adve, 2002). LLVM intermediate language is assembly-like language with simple RISC-like instructions. It provides easy construction of control flow graphs of program functions and of entire programs. There is a number of tools using LLVM for various purposes, including software verification. LLVM has front-ends for C, C++, Ada and Fortran, while there are external projects for translating a number of other languages to LLVM intermediate representation (e.g., Python, Ruby, Haskell, Java, D, PHP, Pure, and Lua).

Software verification and LAV. Verification of software and automated bug finding are some of the greatest challenges in computer science. Software bugs cost the world economy billions of dollars annually (Tassey, 2002). Software verification

¹ <http://argo.matf.bg.ac.rs/?content=lav>

² <http://llvm.org/>

tools aim at automatically checking correctness properties. Different approaches to automated checking of software properties exist, such as symbolic execution (King, 1976), model checking (Clarke, 2008) and abstract interpretation (Cousot & Cousot, 1977). Software verification tools usually use automated theorem provers.

LAV (Vujošević-Janičić & Kuncak, 2012) is an open-source tool for statically verifying program assertions and locating bugs such as buffer overflows, pointer errors and division by zero. LAV uses popular LLVM infrastructure. As a result, it supports several programming languages that compile into LLVM, and benefits from the robust LLVM front ends. LAV is primarily aimed at programs in the C programming language, in which the opportunities for errors are abundant. For each safety critical command, LAV generates a first order logic formula that represents its correctness condition. This formula is checked by one of the several SMT solvers (Barrett, Sebastiani, Seshia, & Tinelli, 2009) used by LAV. If a command cannot be proved to be safe, LAV translates a potential counterexample from the solver into a program trace that exhibits this error. It also extracts the values of relevant program variables along this trace. LAV was already used, to a limited extent, for automated bug finding in students' assignments (Vujošević-Janičić & Kuncak, 2012).

Control flow graph. A control flow graph (CFG) is a graph-based representation of all paths that might be traversed through a program during its execution. Each node of CFG represents a sequence of commands containing only one path of execution (there are no jumps, loops, conditional statements, etc.). The control flow graphs can be produced by various tools, including LLVM. A control flow graph clearly separates the structure of the program and its contents. Therefore, it is a suitable representation for structural comparison of programs.

Graph similarity and neighbor matching method. There are many similarity measures for graphs and their nodes (Kleinberg, 1999; Heymans & Singh, 2003; Blondel, Gajardo, Heymans, Snellart, & van Dooren, 2004; Nikolić, 2013). These measures have been successfully applied in several practical domains like ranking of query results, synonym extraction, database structure matching, construction of phylogenetic trees, analysis of social networks, etc. A short overview of similarity measures for graphs can be found in the literature (Nikolić, 2013).

A specific similarity measure for graph nodes called *neighbor matching*, possesses properties relevant for our purpose that other similar measures lack (Nikolić, 2013). It allows similarity measure for graphs to be defined based on similarity scores of their nodes. The notion of similarity of nodes is based on the intuition that *two nodes i and j of graphs A and B are considered to be similar if neighbor nodes of i can be matched to similar neighbor nodes of j* . More detailed definitions follow.

In the neighbor matching method, if a graph contains an edge (i, j) , the node i is called an *in-neighbor* of node j in the graph and the node j is called an *out-neighbor* of the node i in the graph. An *in-degree* $id(i)$ of the node i is the number of in-neighbors of i , and an *out-degree* $od(i)$ of the node i is the number of out-neighbors of i .

If A and B are two finite sets of arbitrary elements, a *matching* of elements of sets A and B is a set of pairs $M = \{(i, j) | i \in A, j \in B\}$ such that no element of one set is paired with more than one element of the other set. For the matching M , *enumeration functions* $f : \{1, 2, \dots, k\} \rightarrow A$ and $g : \{1, 2, \dots, k\} \rightarrow B$ are defined

such that $M = \{(f(l), g(l)) | l = 1, 2, \dots, k\}$ where $k = |M|$. If $w(a, b)$ is a function assigning weights to pairs of elements $a \in A$ and $b \in B$, the *weight of a matching* is the sum of weights assigned to the pairs of elements from the matching. The goal of the *assignment problem* is to find a matching of elements of A and B of the highest weight (if two sets are of different cardinalities, some elements of the larger set will not have corresponding elements in the smaller set). The assignment problem is usually solved by the well-known Hungarian algorithm of complexity $O(mn^2)$ where $m = \max(|A|, |B|)$ and $n = \min(|A|, |B|)$ (Kuhn, 1955), but there are also more efficient algorithms.

The calculation of similarity of nodes i and j , denoted x_{ij} , is based on iterative procedure given by the following equations:

$$x_{ij}^{k+1} \leftarrow \frac{s_{in}^{k+1}(i, j) + s_{out}^{k+1}(i, j)}{2}$$

where

$$s_{in}^{k+1}(i, j) \leftarrow \frac{1}{m_{in}} \sum_{l=1}^{n_{in}} x_{f_{ij}^{in}(l)g_{ij}^{in}(l)}^k \quad s_{out}^{k+1}(i, j) \leftarrow \frac{1}{m_{out}} \sum_{l=1}^{n_{out}} x_{f_{ij}^{out}(l)g_{ij}^{out}(l)}^k \quad (1)$$

$$\begin{aligned} m_{in} &= \max(id(i), id(j)) & m_{out} &= \max(od(i), od(j)) \\ n_{in} &= \min(id(i), id(j)) & n_{out} &= \min(od(i), od(j)) \end{aligned}$$

where functions f_{ij}^{in} and g_{ij}^{in} are the enumeration functions of the optimal matching of in-neighbors for nodes i and j with weight function $w(a, b) = x_{ab}^k$, and analogously for f_{ij}^{out} and g_{ij}^{out} . In Equations 1, $\frac{0}{0}$ is defined to be 1 (used in case when $m_{in} = n_{in} = 0$ or $m_{out} = n_{out} = 0$). Initial similarity values x_{ij}^0 are set to 1 for each i and j . The termination condition is $\max_{ij} |x_{ij}^k - x_{ij}^{k-1}| < \varepsilon$ for some chosen precision ε and the iterative algorithm is proved to converge (Nikolić, 2013).

The similarity matrix $[x_{ij}]$ reflects the similarities of nodes of two graphs A and B . The similarity of the graphs can be defined as the weight of the optimal matching of nodes from A and B divided by the number of matched nodes (Nikolić, 2013).

3 The Need for Synergy of Testing, Verification, and Similarity Measurement

Automated testing of programs is a very important part of the evaluation process. Unfortunately, the grading system is directly influenced by the choice of test cases. Also, no matter whether the test cases are automatically generated or manually designed, testing cannot guarantee neither functional correctness of a program nor the absence of bugs.

For checking functional correctness, combination of random testing with evaluator-supplied test cases is a common choice (Mandal, Mandal, & Reade, 2007). However, randomly generated test cases are not likely to hit a bug if it exists (Godefroid, Levin, & Molnar, 2012), while manually choosing all important test cases is not a trivial job and can be time consuming. It is not sufficient that test cases cover all important paths through the program. It is also important to

carefully choose values of the variables for each path — for some values along the same path a bug can be detected while for some other values the bug can stay undetected.

Also, manually generated test cases are designed according to the expected solutions, while the evaluator cannot predict all the important paths through the student’s solution. Even running a test case that hits a certain bug (for example, a buffer overflow bug in a C program) does not necessarily lead to any visible undesired behavior if the running is done in a normal (or sandbox) environment. Finally, if one manages to hit a bug by a test case, if the bug produces the *Segmentation fault* message, it is not a feedback that student can easily understand and use for debugging the program. In the context of automated grading, this feedback cannot be easily used since it may have different causes. In contrast to program testing, software verification tools like Pex (Tillmann & Halleux, 2008), Klee (Cadaru, Dunbar, & Engler, 2008), S2E (Chipounov, Kuznetsov, & Candea, 2011), CBMC (Clarke, Kroening, & Lerda, 2004), ESBMC (Cordeiro, Fischer, & Marques-Silva, 2009), and LAV (Vujošević-Janičić & Kuncak, 2012) can give much better explanations (e.g., the kind of bug and the program trace that introduces an error).

```

0: #define max_size 50
1: void matrix_maximum(int a[][max_size], int rows, int columns, int b[])
2: {
3:     int i, j, max=a[0][0];                int i, j, max;
4:     for(i=0; i<rows; i++)                  for(i=0; i<rows; i++)
5:     {                                       {
6:                                           max = a[i][0];
7:         for(j=0; j<columns; j++)          for(j=0; j<columns; j++)
8:             if(max < a[i][j])              if(max < a[i][j])
9:                 max = a[i][j];              max = a[i][j];
10:            b[i] = max;                      b[i] = max;
11:            max=a[i+1][0];
12:        }                                   }
13:    return;                                return;
14: }
```

Fig. 1 Buffer overflow in the code on left-hand side cannot be discovered by simple testing. Functionally equivalent solution without a bug is given on right-hand side.

The example function shown at Figure 1 is extracted from a student’s code written on an exam. It calculates the maximum value of each row of a matrix and writes these values into an array. This function is used in a context where the memory for the matrix is statically allocated and numbers of rows and columns are less or equal to the allocated sizes of the matrix. However, in the line 11, there is a possible buffer overflow bug, since $i + 1$ can exceed the allocated number of rows for the matrix. It is possible that this kind of a bug does not affect the output of the program or destroy any data, but in a slightly different context it can be harmful, so students should be warned and penalized for making such errors. The bugs like this one can be missed in testing but are easily discovered by verification tools like LAV.

Functional correctness and absence of bugs are not the only important aspects of students’ programs. The programs are often supposed to meet certain require-

ments concerning the structure of the program, such as its modularity (adequate decomposition of code to functions) or simplicity. Figure 2 shows two solutions of different modularity or structural simplicity for two problems. Neither testing, nor software verification can be used to assess these aspects of the programs. This problem can be addressed by checking the similarity of student's solution with a teacher provided solution, i.e., by analyzing the similarity of their related graphs (e.g. CFGs) (Wang et al., 2007; Naudé et al., 2010; Nikolić, 2013).³

Problem	First solution	Second solution
1.	<pre>if(a<b) n = a; else n = b; if(c<d) m = c; else m = d;</pre>	<pre>n = min(a, b); m = min(c, d);</pre>
2.	<pre>for(i=0; i<n; i++) for(j=0; j<n; j++) if(i==j) m[i][j] = 1;</pre>	<pre>for(i=0; i<n; i++) m[i][i] = 1;</pre>

Fig. 2 Examples extracted from two students' solutions of the same problem

Finally, using similarity only (like in (Wang et al., 2007; Naudé et al., 2010)) or even with support of a bug finding tool, would miss to penalize incorrectness of program's behavior. Figure 3 gives a simple example program, extracted from a real student's solution, that is very similar to the expected solution and without verification errors. However, this program is not functionally correct. Therefore, we conclude that the synergy of these three approaches is needed for sophisticated evaluation of students' assignments.

<pre>max = 0; for(i=0; i<n; i++) if(a[i] > max) max = a[i];</pre>	<pre>max = a[0]; for(i=1; i<n; i++) if(a[i] > max) max = a[i];</pre>
---	--

Fig. 3 Code extracted from student's solution (left-hand side) and expected solution (right-hand side). In the student's solution there are no verification bugs, it is very similar to the expected solution but it does not perform the desired behavior (in the case when all elements of the array *a* are negative integers).

³ In Figure 2, the second example could also be distinguished by profiling for large inputs, because it is quadratic in one case and linear in the other. However, profiling cannot be used to assess structural properties in general.

4 Grading Setting

There may be different grading settings depending on aims of the course and goals of teachers. The setting used at an introductory course of programming in C (at University of Belgrade) is taking exams on computers and expecting from students to write working programs. In order to help students achieve this goal, each assignment is provided with several test cases which illustrate desired behavior of the solution. Students are also provided with sufficient (but limited) time for developing and testing programs. If a student fails to provide a working program that gives correct results for given test cases, his/her solution is not further examined. Otherwise, the program is tested by additional test cases (unknown to students) and a certain amount of points is given corresponding to the test cases successfully passed. Only if all these test cases are successfully passed, the program is further manually examined and may obtain additional points with respect to other features of the program (efficiency, modularity, simplicity, absence of bugs, etc).

All experiments described in this paper were performed on a corpus of programs written by students on the exams, following the described grading setting. The corpus consists of 266 solutions to 15 different problems. These problems include numerical calculations, manipulations with arrays and matrices, manipulations with strings, and manipulations with data structures. Only programs that passed all test cases were included in this corpus. These programs are the main target of our automated evaluation technique since the manual grading was applied only in this case and we want to explore potentials for completely eliminating manual grading. These programs obtained 80% of the maximal score (as they passed all test cases) and additional potential 20% were given by manual inspection. The grades are expressed at the scale from 0 to 10. The corpus together with problem descriptions and the final marks are publicly available.⁴

5 Assignment Evaluation and Software Verification

In this section we show benefits of using software verification tool in assignment evaluation, e.g., generating useful feedback for students and providing improved assignment evaluation for teachers.

5.1 Software verification for assignment evaluation

No software verification tool can report all the bugs in a program without introducing false alarms (due to the undecidability of the halting problem). False alarms (i.e., reported "bugs" that are not real bugs) arise as a consequence of approximations that are necessary in modeling of programs.

The most important approximation is concerned with dealing with loops. Different verification approaches use various techniques for dealing with loops. These techniques range from under-approximations of loops to over-approximations of loops. Under-approximation of loops, as in bounded model checking techniques

⁴ <http://argo.matf.bg.ac.rs/?content=lav>

(Clarke, 2008), uses a fixed number n for loop unwinding. In this case, if the code is verified successfully, it means that the original code has no bugs for n or less passes through the loop. However, it may happen that some bug remains undiscovered if the unwinding is performed an insufficient number of times. Over-approximation of loops can be done by simulation of first n and last m passes through the loop (Vujošević-Janičić & Kuncak, 2012) or by using abstract interpretation techniques (Cousot & Cousot, 1977). If there are no bugs detected in the over-approximated code, then the original code has no bugs too. However, in this case, a false alarm can appear after or inside a loop. On the other hand, precise dealing with loops, like in symbolic execution techniques, can be non terminating.

False alarms are highly unwelcome in software development, but still are not critical — the developer can fix the problem or confirm that the reported problem is not really a bug (and both of these are situations that the developer can expect and understand). However, false alarms in assignment evaluation are rather critical and have to be eliminated. For teachers, there should be no false alarms, because the evaluation process should be as automatic and reliable as possible. For students, there should be no false alarms because they would be confused if told that something is a bug when it is not. In order to eliminate false alarms, a system may be non-terminating or may miss to report some real bugs. In assignment evaluation, the second choice is more reasonable — the tool has to be terminating, must not introduce false alarms, even if the price is missing some real bugs. These requirements make applications of software verification in education rather specific, and special care has to be taken when these techniques are applied.

5.2 LAV for assignment evaluation

LAV is a general purpose verification tool and has a number of options that can adapt its behavior to the desired context. When running LAV in the assignment evaluation context, most of these options can be fixed.

The most important choice for the user is the choice of the way in which LAV deals with loops. LAV has support for both over-approximation of loops and for fixed number of unwinding of loops (under-approximation), two common techniques for dealing with loops. Setting up the upper loop bound (if under-approximation is used), is problem dependent and should be done by the teacher for each assignment.

We use LAV in the following way. LAV is first invoked with its default parameters — over-approximation of loops. Since this technique can introduce false alarms, if a potential bug is found after or inside a loop, the verification is invoked again but this time with fixed unwinding parameter. If the bug is still present, then it is reported. Otherwise, the previously detected potential bug is considered to be a false alarm and it is not reported.

In software verification, each detected bug is important and should be reported. However, some bugs can confuse novice programmers, like the one shown in Figure 4. In this code, at the line 11, there is a possible buffer overflow. For instance, for $n = 0x80000001$ only 4 bytes will be allocated for the pointer `array`, because of an integer overflow. This is a verification error, but a teacher may decide not to consider this kind of bugs. For this purpose, LAV can be invoked in mode for students (so the bugs like this one are not reported).

```

1: unsigned i, n;
2: unsigned *arr;
3: scanf("%u", &n);
4: array = malloc(n*sizeof(unsigned));
5: if(array == NULL)
6: {
7:     fprintf(stderr, "Unsuccessful allocation\n");
8:     exit(EXIT_FAILURE);
9: }
10: for(i=0; i<n; i++)
11:     array[i] = i;

```

Fig. 4 Buffer overflow in this code is a verification error, but the teacher may decide not to consider this kind of bugs.

To a limited extent, LAV was already used on students' assignments at an introductory programming course (Vujošević-Janičić & Kuncak, 2012). In these experiments, most of the programs from the corpus were not functionally correct. It was shown that the vast majority of bugs, produced by students, follow wrong expectations — for instance, expectations that input parameters of their programs will meet certain constraints and that memory allocation will always succeed. It is also noticed that most of the reported bugs are consequence of only few oversights. In many cases, omission of a necessary check produces several bugs in the rest of the program. Therefore, the number of bugs, as reported by a verification tool, is not a reliable indicator of program quality. This property will be taken into account in automated grading.

5.3 Experimental evaluation

As discussed in Section 3, programs that successfully pass a testing phase can still contain bugs. To show that this problem is practically important, we used LAV to analyze programs from the corpus described in Section 4.

For each problem, LAV was ran with its default parameters, and programs with potential bugs were checked with under-approximation of loops, as described in Section 5.2.⁵ The results are shown in Table 1. The time that LAV spent in analyzing the programs was typically negligible.⁶ LAV discovered bugs in 35 solutions that successfully passed the testing. There was one bug missed by manual inspection and detected by LAV and one bug missed by LAV and detected by manual inspection. The bug missed by manual inspection was the one described in Section 3 and given in Figure 1. The bug missed by LAV was a consequence of the problem formulation which was too general to allow a precise unique upper

⁵ When analyzing the solutions of problems 3, 5 and 8, only under-approximation of loops was used. This was the consequence of the formulation of the problems given to the students. Namely, the formulation of these problems contained some assumptions on input parameters. These assumptions implied that some potential bugs should not be considered (because these are not bugs when these additional assumptions are taken into account).

⁶ Generally, in this context, a time limit can be given to the verification tool and if it was exceeded no bug will be reported (in order to avoid reporting false alarms) or a program can be checked using the same parameters but with another underlying solver (if applicable for the tool).

loop unwinding parameter value for all possible solutions. There were just two false alarms produced by LAV when the default parameters were used. These false alarms were eliminated when the tool was invoked for the second time with a specified loop unwinding parameter, and hence there were no false alarms in the final outputs. In summary, the presented results show that a verification tool like LAV can be used as a complement to automated testing that improves the evaluation process.

Table 1 Summary of bugs in the corpus: the second column represents the number of students' solutions to the given problem; the third and the fourth column represents the number of solutions with bugs detected by manual inspection and by LAV; the fifth column gives the number of programs shown to be bug-free by LAV (over/under approximation); the sixth column gives the number of false alarms made by LAV invoked with default parameters and, if applicable, with under-approximation.

problem	# solutions	# programs with bugs by manual inspection	# programs with bugs by LAV	# bug-free programs by LAV def./custom	# false alarms with def./custom parameters
1.	44	0	0	44/-	0/-
2.	32	11	11	20/1	1/0
3.	7	2	2	-/5	-/0
4.	5	0	1	3/1	1/0
5.	12	3	2	-/10	-/0
6.	7	0	0	6/1	1/0
7.	33	0	0	33/-	0/-
8.	31	11	11	-/20	-/0
9.	10	6	6	4/0	0/0
10.	14	2	2	12/0	0/0
11.	31	0	0	31/-	0/-
12.	18	0	0	18/-	0/-
13.	3	0	0	3/-	0/-
14.	7	0	0	7/-	0/-
15.	12	0	0	12/-	0/-
total	266	35	35	193/38	2/0

5.4 Feedback for students and teachers

LAV can be used to provide a meaningful and comprehensible feedback to students while writing their programs. Information like the line number, the kind of the error, program trace that introduces the error and values of the variables along this trace, can help student improve the solution. It can also remind the student to add an appropriate check that is missing. The example given in Figure 5, extracted from a student's code written on an exam, shows the error detected by LAV and the generated hint.

From the software verification support, a teacher can obtain the information if the student's program contains a bug. The teacher can use this information in grading assignments by himself. Alternatively, this information can be taken into account within the wider integrated framework for obtaining automatically proposed final grade, as discussed in Section 7.

<pre> 1: #include<stdio.h> 2: #include<stdlib.h> 3: int get_digit(int n, int d); 4: int main(int argc, char** argv) 5: { 6: int n, d; 7: n = atoi(argv[1]); 8: d = atoi(argv[2]); 9: printf("%d\n", get_digit(n, d)); 10: return 0; 11: }</pre>	<pre> verification failed: line 7: UNSAFE function: main error: buffer_overflow in line 7: counterexample: argc == 1, argv == 1 HINT: A buffer overflow error occurs when trying to read or write outside the reserved memory for a buffer/array. Check the boundaries of the array!</pre>
--	--

Fig. 5 Listing extracted from student’s code written on an exam (left-hand side) and LAV’s output (right-hand side)

6 Assignment Evaluation and Structural Similarity of Programs

In this section we propose a similarity measure for programs based on their control flow graphs, perform its experimental evaluation, and point to ways it can be used to provide feedback for students and teachers.

6.1 Similarity of CFGs for assignment evaluation

To evaluate structural properties of programs, we take the approach of comparing students’ programs to solutions provided by the teacher. Student’s program is considered to be good if it is similar to some of the programs provided by the teacher (Wang et al., 2007). In order to perform a comparison, a suitable program representation and a similarity measure are needed. As already noticed in Section 2, there is a control flow graph (CFG) corresponding to each program. The CFG reflects the structure of the program. Also, there is a linear code sequence attributed to each node of the CFG which we call the node content. We assume that the code is in the intermediate LLVM language. In order to measure the similarity of programs, both the similarity of graphs’ structures and the similarity of node contents should be considered. We take the approach of combining the similarity of node contents with topological similarity of graph nodes described in Section 2.

Similarity of node contents. The node content is a sequence of LLVM instructions. A simple way of measuring the similarity of two sequences of instructions s_1 and s_2 is using the edit distance between them $d(s_1, s_2)$ — the minimal number of insertion, deletion and substitution operations over the elements of the sequence by which one sequence can be transformed into another (Levenshtein, 1966). In order for edit distance to be computed, the cost of each insertion, deletion and substitution operation has to be defined. We define the cost of insertion and deletion of an instruction to be 1. Next, we define the cost of substitution of instruction i_1 by instruction i_2 . Let *opcode* be a function that maps an instruction to its opcode (a part of instruction that specifies the operation to be performed). Let *opcode*(i_1) and *opcode*(i_2) be function calls. Then, the cost of substitution is 1 if i_1 and i_2 call different functions, and 0 if they call the same function. If *opcode*(i_1) or *opcode*(i_2)

is not a function call, the cost of substitution is 1 if $opcode(i_1) \neq opcode(i_2)$, and 0 otherwise. Let $n_1 = |s_1|$, $n_2 = |s_2|$, and let M be the maximal edit distance over two sequences of length n_1 and n_2 . Then, the similarity of sequences s_1 and s_2 is defined as $1 - d(s_1, s_2)/M$.

Although it could be argued that the proposed similarity measure is rough since it does not account for differences of instruction arguments, it is simple, easily implemented, and intuitive.

Full similarity of nodes and similarity of CFGs. The topological similarity of nodes can be computed by the method described in Section 2. However, purely topological similarity does not account for differences of the node content. Hence, we modify the computation of topological similarity to include the apriori similarity of nodes. The modified update rule is:

$$x_{ij}^{k+1} \leftarrow \sqrt{y_{ij} \cdot \frac{s_{in}^{k+1}(i, j) + s_{out}^{k+1}(i, j)}{2}}$$

where y_{ij} are the similarities of contents of nodes i and j and $s_{in}^{k+1}(i, j)$ and $s_{out}^{k+1}(i, j)$ are defined by Equations 1. Also, we set $x_{ij}^0 = y_{ij}$. This way, both content similarity and topological similarity of nodes are taken into consideration. The similarity of CFGs can be defined based on the node similarity matrix as described in Section 2. Note that both the similarity of nodes and the similarity of CFGs take values in the interval $[0, 1]$.

It should be noted that our approach provides both the similarity measure for CFGs and the similarity measure for their nodes (x_{ij}). In addition to evaluating similarity of programs, this approach enables matching of related parts of the programs by matching the most similar nodes of CFGs. This could serve as a basis of a method for suggesting which parts of the student's program could be further improved.

6.2 Experimental evaluation

In order to show that the proposed program similarity measure corresponds to some intuitive notion of program similarity, we performed the following experiment. For each program from the corpus already described in Section 4, we found the most similar program from the rest of the corpus and counted how often these programs are the solutions for the same problem. That was the case for 90% of all programs. This shows that our similarity measure performs well since with high probability, for each program, the program that is the most similar to it, corresponds to the same problem. The inspection suggests that in most cases, where the programs do not correspond to the same problem, student took an original approach to solving the problem.

The CFGs of the programs from the corpus are rather small. The average size of CFGs is 15 nodes. The time spent to compute the similarity of two programs is negligible. However, out of the educational context where CFGs could have thousands of nodes, the scalability might be an issue.

6.3 Feedback for students and teachers

The students can benefit from program similarity evaluation while learning and exercising, assuming that the teacher provided a valid solution or set of solutions to the evaluation system. In introductory programming courses, most often a student's solution can be considered as better if it is more similar to one of the teacher's solutions (Wang et al., 2007). In Section 7 we show that the similarity measure can be used for automatic calculation of a grade (a feedback that students easily understand). Moreover, we show that there is a significant linear dependence of the grade on the similarity value. Due to that linearity, the similarity value can be considered as an intuitive feedback, but also it can be translated into descriptive estimate. For example, the feedback could be that the solution is dissimilar (0-0.5), roughly similar (0.5-0.7), similar (0.7-0.9) or very similar (0.9-1) to one of the desired solutions.

The teachers can use the similarity information in automated grading, as discussed in Section 7.

7 Automated Grading

We believe that automated grading can be performed by calculating a linear combination of different scores measured for the student's solution. We propose a linear model for prediction of the teacher-provided grade of the following form:

$$\hat{y} = \alpha_1 \cdot x_1 + \alpha_2 \cdot x_2 + \alpha_3 \cdot x_3$$

where

- \hat{y} is the automatically predicted grade,
- x_1 is a result obtained by automated testing expressed in the interval $[0, 1]$,
- x_2 is 1 if in the student's solution is correct as reported by the software verification tool, and 0 otherwise,
- x_3 is the maximal value of similarity between the student's solution and each of the teacher provided solutions (its range is $[0, 1]$).

It should be noted that we do not use bug count as a parameter, as discussed in Section 5.2. Different choices for the coefficients α_i , for $i = 1, 2, 3$ could be proposed. In our case, one simple way could be $\alpha_1 = 8$, $\alpha_2 = 1$, and $\alpha_3 = 1$ since all programs in our training set won 80% of the full grade due to the success in testing. However, it is not always clear how the teacher's intuitive grading criterion can be factored to automatically measurable quantities. Teachers need not have the intuitive feeling for all the variables involved in the grading. For instance, the behavior of any of the proposed similarity measures including ours (Wang et al., 2007; Naudé et al., 2010; Nikolić, 2013) is not clear from their definitions only. So, it may be unclear how to choose weights for different variables when combining them in the final grade or if some of the variables should be nonlinearly transformed in order to be useful for grading. A natural solution is to try to tune the coefficients α_i , for $i = 1, 2, 3$ so that the behavior of the predictive model corresponds to the teacher's grading style. For that purpose, coefficients can be determined automatically using least squares linear regression (Gross, 2003) if a manually graded corpus of students' programs is provided by the teacher.

In our evaluation the corpus of programs was split into a training and a test set where the training set consisted of two thirds of the corpus and the test set consisted of one third of the corpus. The training set contained solutions of eight different problems and the test set contained solutions of remaining seven problems.

Due to the nature of the corpus, for all the instances it holds $x_1 = 1$. Therefore, while it is clear that the number of test cases the program passed (x_1) is useful in automated grading, this variable can not be analyzed based on this corpus.

The optimal values of coefficients α_i , $i = 1, 2, 3$, with respect to the training corpus, are determined using least squares linear regression. The obtained equation is

$$\hat{y} = 6.058 \cdot x_1 + 1.014 \cdot x_2 + 2.919 \cdot x_3$$

The formula for \hat{y} may seem counterintuitive. Since the minimal grade in the corpus is 8 and $x_1 = 1$ for all instances, one would expect that it holds $\alpha_1 \approx 8$. The discrepancy is due to the fact that for the solutions in the corpus, the minimal value for x_3 is 0.68 — since the solutions are good (they all passed the testing) there are no programs with low similarity value. Taking this into consideration, one can rewrite the formula for \hat{y} as

$$\hat{y} = 8.043 \cdot x_1 + 1.014 \cdot x_2 + 0.934 \cdot x'_3$$

where $x'_3 = \frac{x_3 - 0.68}{1 - 0.68}$ so the variable x'_3 takes values from the interval $[0, 1]$. This means that when the range of variability of both x_2 and x_3 is scaled to the interval $[0, 1]$, their contribution to the mark is rather similar.

Table 2 shows the comparison between the model \hat{y} and three other models. Model $\hat{y}_1 = 8 \cdot x_1 + x_2 + x_3$ has predetermined parameters, model \hat{y}_2 is trained just with verification information x_2 (without similarity measure), and model \hat{y}_3 is trained only with similarity measure x_3 (without verification information). Results show that the performance of model \hat{y} on the test set (consisting of problems not appearing in the training set) is outstanding — the correlation is 0.842 and the model accounts for 71% of the variability of teacher provided grade. These results indicate a strong and reliable dependence between teacher provided grade and the variables x_i , meaning that a grade can be reliably predicted by \hat{y} . Also, \hat{y} is much better than other models. This shows that the approach using both verification information and graph similarity information is superior to approaches using only one source of information, and also that automated tuning of coefficients of the model provides better prediction than giving them in advance.

Inspection of solutions that yielded the biggest error in prediction suggests that the greatest source of discrepancy of automatically provided and teacher provided grades are the original solutions given by students and the solutions that the teacher did not predict in advance. However, we cannot exclude other factors apart from presence of bugs and similarity to model solutions, that govern human grading process.

8 Related work

Automated testing is the most common way of evaluating students' programs (Douce et al., 2005). Test cases are usually supplied by a teacher and/or randomly generated (Mandal et al., 2007). A lot of systems use this approach, for

	r	$r^2 \cdot 100\%$	Rel. error
\hat{y}	0.842	71%	10.1%
\hat{y}_1	0.730	53.3%	12.8%
\hat{y}_2	0.620	38.4%	16.7%
\hat{y}_3	0.457	20.9%	17.7%

Table 2 The performance of the predictive model on the training and test set. We provide correlation coefficient (r), the fraction of variance of y accounted by the model ($100 \cdot r^2$), and relative error — average error divided by the length of the range in which the grades vary (which is 8 to 10 in the case of this particular corpus).

example, PSGE (Hext & Winings, 1969), Kassandra (Matt, 1994), BOSS (Joy, Griffiths, & Boyatt, 2005), WebToTeach (Arnow & Barshay, 1999), Schemerobe (Saikkonen, Malmi, & Korhonen, 2001), TRY (Jones, 2001), HoGG (Morris, 2002), BAGS (Morris, 2003), on-line Judge (Cheang, Kurnia, Lim, & Oon, 2003), JEWL (English, 2004), Quiver (Ellsworth, Fenwick, & Kurtz, 2004), and JUnit (Wick, Stevenson, & Wagner, 2005).

Software verification techniques are not commonly used in automated evaluation of programs. There are limited experiments on using Java PathFinder model checker for automated test case generation (Ihantola, 2007). Tools with integrated support for automated testing and verification, e.g. Ceasar (Garavel, 1998), are usually too complex and not aimed for educational purposes. To the authors’ knowledge, there is no other software verification tool deployed in process of automated bug finding as a complement to automated testing of students’ programs. The tool LAV was already used, to a limited extent, for finding bugs in students’ programs (Vujošević-Janičić & Kuncak, 2012). In that work, a different sort of corpus was used, as discussed in Section 5.2. Also, that application did not aim at automated grading, and instead was made in the wider context of design and development of LAV as a general-purpose SMT-based error finding platform.

Wang et al. proposed a grading approach for assignments in C based only on program similarity (Wang et al., 2007). It relies on dependence graphs (Horwitz & Reps, 1992) as program representation. They perform various code transformations in order to standardize the representation of the program. In this approach, the similarity is calculated based on comparison of structure, statement, and size which are weighted by some predetermined coefficients. Their approach is evaluated on 10 problems, 200 solutions each, and obtain good results compared to manual grading. Manual grading was performed strictly according to the criterion that indicates how the scores are awarded for structure, statements used, and size. However, it is not quite obvious that human grading is always expressed strictly in terms of these three factors. An advantage of our approach compared to this one is automated tuning of weights corresponding to different variables used in grading, instead of using the predetermined ones. Since teachers do not need to have an intuitive feeling for different similarity measures, it may be unclear how the corresponding weights should be chosen. Also, we avoid language dependent transformations by using LLVM which makes our approach applicable to large variety of programming languages. Very similar approach to the one of Wang et al. was presented by Li et al. (Li, Pan, Zhang, Chen, Nie, & He, 2010).

Another approach to grading assignments based only on graph similarity measure is proposed by Naudé et al. (Naudé et al., 2010). They represent programs

as dependence graphs and propose directed acyclic graph (DAG) similarity measure. In their approach, for each solution to be graded, several similar solutions in the training set are found and the grade is formed by combining grades of these solutions with respect to matched portions of the similar solutions. The approach was evaluated on one assignment problem and the correlation between human and machine provided grades is the same as ours. For appropriate grading they recommend at least 20 manually graded solutions of various qualities for each problem to be automatically graded. In the case of automatic grading of high quality solutions (as is the case with our corpus), using 20 manually graded solutions, their approach achieves 16.7% relative error, while with 90 manually graded solutions it achieves around 10%. The improvement that our approach provides is reflected through several indicators. We used a heterogeneous corpus of 15 problems instead of one. Our approach uses 1 to 3 model solutions for each problem to be graded and a training set for weight estimation which does not need to contain the solutions for the program to be graded. So, after the initial training has been performed, for each new problem only few model solutions should be provided. Using 1 to 3 model solutions, we achieve 10% relative error (see Table 2). Due to the use of the LLVM platform, we do not use language dependent transformations, so our approach is applicable to large number of programming languages. The similarity measure we use, called neighbor matching, is similar to the one of Naudé et al., but for our measure, important theoretical properties (e.g. convergence) are proven (Nikolić, 2013). The neighbor matching method was already applied to several problems but in all these applications its use was limited to ordinary graphs with nodes without any internal specifics. In order to be applied to CFGs, the method was modified to include node content similarity which was independently defined as described in Section 6.1.

Finally, as a distinctive feature of our system, we are not aware of open source implementations of the similarity based approaches. A drawback in the comparison of our approach to previously described ones is that our corpus consists of high quality solutions due to the grading setting at the course.

Apart of assignment grading, regression techniques were also used for final grade forecasting with good results. For this purpose, Macfadyen et al. used data from learning management system and identified variables most useful for the prediction, e.g., number of assessments completed and number of discussion and mail messages sent (Macfadyen & Dawson, 2010). Kotsiantis performed successful forecasting based on demographic characteristics of students, results of several written assignments, and class attendance (Kotsiantis, 2012).

9 Conclusions and Further Work

We presented two techniques that can be used for improving automated evaluation of students' programs. First one is based on software verification and second one on CFG similarity measurement. Both techniques can be used for providing useful and helpful feedback to students and for improving automated grading for teachers. In our evaluation, we show that synergy of these techniques offers more information useful for automated grading than any of them independently. Also, we obtained good results in prediction of the grades for a new set of assignments. This shows that our approach can be trained to adapt to teacher's grading style on

several teacher graded problems and then be used on different problems using only few model solutions per problem. An important advantage of our approach is independence of specific programming language since LLVM platform (which we use to produce intermediate code) supports large number of programming languages. We also provide the corresponding open source tools.

In our future work we are planning to make an integrated web-based system with support for the mentioned techniques along with compiling, automated testing, profiling and detection of plagiarism of students' programs. Also, we intend to improve feedback to students by indicating missing or redundant parts of code compared to the teacher's solution. This feature would rely on the fact that our similarity measure provides the similarity values for nodes of CFGs, and hence enables matching the parts of code between two solutions. If some parts of the solutions cannot be matched or are matched with very low similarity, this can be reported to the student. On the other hand, the similarity of the CFG with itself could reveal the repetitions of parts of the code and suggest that refactoring could be performed. We are planning to integrate LLVM-based open source tool KLEE (Cadarc et al., 2008) for automated test case generation and also to add support for teacher supplied test cases.

We are also planning to explore potential for using software verification tools for proving functional correctness of students' programs. This task would pose new challenges. Testing, profiling, bug finding and similarity measurement are used on original students' programs, which makes the automation easy. For verification of functional correctness, the teacher would have to define correctness conditions (possibly in terms of implemented functions) and insert corresponding assertions in appropriate places in students' programs which should be possible to automate in some cases, but it is not trivial in general. In addition, for some programs it is not easy to formulate correctness conditions (for example, for programs that are expected only to print some messages on standard output).

References

- Ala-Mutka, K. M. (2005). A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, 15, 83–102.
- Allen, I. E., & Seaman, J. (2010). Learning on demand: Online education in the united states, 2009. Tech. rep., The Sloan Consortium.
- Arnold, D., & Barshay, O. (1999). Webtoteach: an interactive focused programming exercise system. *Frontiers in Education, Annual*, 1, 12A9/39–12A9/44vol.1.
- Barrett, C., Sebastiani, R., Seshia, S. A., & Tinelli, C. (2009). Satisfiability modulo theories. In *Handbook of Satisfiability*, Vol. 185 of *Frontiers in Artificial Intelligence and Applications*, pp. 825–885. IOS Press.
- Blondel, V. D., Gajardo, A., Heymans, M., Snellart, P., & van Dooren, P. (2004). A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM Review*, 46, 647–666.
- Cadarc, C., Dunbar, D., & Engler, D. (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceeding OSDI'08 Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association Berkeley.

- Cheang, B., Kurnia, A., Lim, A., & Oon, W.-C. (2003). On automated grading of programming assignments in an academic institution. *Comput. Educ.*, 41(2), 121–131.
- Chipounov, V., Kuznetsov, V., & Candea, G. (2011). S2e: a platform for in-vivo multi-path analysis of software systems. *SIGARCH Comput. Archit. News*, 39, 265–278.
- Clarke, E., Kroening, D., & Lerda, F. (2004). A tool for checking ansi-c programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176. Springer.
- Clarke, E. M. (2008). *25 Years of Model Checking — The Birth of Model Checking*, Vol. 5000/2008, 1–26 of *Lecture Notes in Computer Science*. Springer.
- Cordeiro, L., Fischer, B., & Marques-Silva, J. (2009). Smt-based bounded model checking for embedded ansi-c software. *International Conference on Automated Software Engineering*, 0, 137–148.
- Cousot, P., & Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pp. 238–252.
- Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3), 4+.
- Ellsworth, C. C., Fenwick, Jr., J. B., & Kurtz, B. L. (2004). The quiver system. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, pp. 205–209, New York, NY, USA. ACM.
- English, J. (2004). Automated assessment of gui programs using jewl. *SIGCSE Bull.*, 36(3), 137–141.
- Garavel, H. (1998). Open/cæsar: An open software architecture for verification, simulation, and testing. In Steffen, B. (Ed.), *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, Vol. 1384 of *Lecture Notes in Computer Science*, pp. 68–84, Berlin. Springer Verlag. Full version available as INRIA Research Report RR-3352.
- Godefroid, P., Levin, M. Y., & Molnar, D. A. (2012). Sage: Whitebox fuzzing for security testing. *ACM Queue*, 10(1), 20.
- Gross, J. (2003). *Linear Regression*. Springer.
- Hext, J. B., & Winings, J. W. (1969). An automatic grading scheme for simple programming exercises. *Commun. ACM*, 12(5), 272–275.
- Heymans, M., & Singh, A. (2003). Deriving phylogenetic trees from the similarity analysis of metabolic pathways. *Bioinformatics*, 19, 138–146.
- Horwitz, S., & Reps, T. (1992). The use of program dependence graphs in software engineering. In *Proceedings of the 14th international conference on Software engineering*, ICSE '92, pp. 392–411, New York, NY, USA. ACM.
- Ihantola, P. (2007). Creating and visualizing test data from programming exercises. *Informatics in education*, 6(1), 81–102.
- Ihantola, P., Ahoniemi, T., Karavirta, V., & Seppälä, O. (2010). Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pp. 86–93, New York, NY, USA. ACM.
- Jones, E. L. (2001). Grading student programs - a software testing approach. *Journal of Computing Sciences in Colleges*, 16(2), 187–194.
- Joy, M., Griffiths, N., & Boyatt, R. (2005). The boss online submission and as-

- essment system. *J. Educ. Resour. Comput.*, 5(3).
- King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, 19(7), 385–394.
- Kleinberg, J. M. (1999). Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46, 604 — 632.
- Kotsiantis, S. B. (2012). Use of machine learning techniques for educational proposes: a decision support system for forecasting students' grades. *Artificial Intelligence Review*, 34(4), 331–344.
- Kuhn, H. W. (1955). The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2), 83–97.
- Lattner, C., & Adve, V. (2002). The LLVM Instruction Set and Compilation Strategy..
- Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8), 707–710.
- Li, J., Pan, W., Zhang, R., Chen, F., Nie, S., & He, X. (2010). Design and implementation of semantic matching based automatic scoring system for c programming language. In *Proceedings of the Entertainment for education, and 5th international conference on E-learning and games*, pp. 247–257. Springer-Verlag.
- Macfadyen, L. P., & Dawson, S. (2010). Mining lms data to develop an "early warning system" for educators: a proof of concept. *Computers and Education*, 54(2), 588–599.
- Mandal, A. K., Mandal, C. A., & Reade, C. (2007). A system for automatic evaluation of c programs: Features and interfaces. *IJWLTT*, 2(4), 24–39.
- Matt, U. V. (1994). Kassandra: The automatic grading system. *SIGCUE Outlook*, 22, 22–26.
- Morris, D. S. (2002). Automatically grading java programming assignments via reflection, inheritance, and regular expressions. *Frontiers in Education Conference 1*, 1, T3G–22.
- Morris, D. (2003). Automatic grading of student's programming assignments: an interactive process and suit of programs. In *Proceedings of the Frontiers in Education Conference 3*, Vol. 3, pp. 1–6.
- Naudé, K. A., Greyling, J. H., & Vogts, D. (2010). Marking student programs using graph similarity. *Computers and Education*, 54(2), 545–561.
- Nikolić, M. (2013). Measuring similarity of graph nodes by neighbor matching. *Intelligent Data Analysis, Accepted for publication*.
- Nipkow, T. (2012). Teaching semantics with a proof assistant: No more lsd trip proofs. In *VMCAI*, pp. 24–38.
- Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., & Paterson, J. (2007). A survey of literature on the teaching of introductory programming. In *Working group reports on ITiCSE on Innovation and technology in computer science education*, ITiCSE-WGR '07, pp. 204–223, New York, NY, USA. ACM.
- Saikkonen, R., Malmi, L., & Korhonen, A. (2001). Fully automatic assessment of programming exercises. *ACM Sigcse Bulletin*, 33, 133–136.
- Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. Tech. rep., National Institute of Standards and Technology.
- Tillmann, N., & Halleux, J. (2008). Pex white box test generation for .net. In *Proc. of TAP 2008, the 2nd International Conference on Tests and Proofs*, Vol.

- 4966 of *LNCS*, pp. 134–153. Springer.
- Vujošević-Janičić, M., & Kuncak, V. (2012). Development and evaluation of LAV: an SMT-based error finding platform. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, LNCS.
- Vujošević-Janičić, M., & Tošić, D. (2008). The role of programming paradigms in the first programming courses. *The Teaching of Mathematics*, *XI*(2), 63–83.
- Wang, T., Su, X., Wang, Y., & Ma, P. (2007). Semantic similarity-based grading of student programs. *Information and Software Technology*, *49*(2), 99–107.
- Wick, M., Stevenson, D., & Wagner, P. (2005). Using testing and junit across the curriculum. *SIGCSE Bull.*, *37*(1), 236–240.