Computer Science and Information Systems 17(1):205–227 https://doi.org/10.2298/CSIS181220019V

Regression Verification for Automated Evaluation of Students Programs

Milena Vujošević Janičić[1] and Filip Marić[1]

University of Belgrade, Faculty of Mathematics, Studentski trg 16
11000 Belgrade, Serbia
{milena,filip}@matf.bg.ac.rs

Abstract. Regression verification is a form of software verification based on formal static analysis of code, which is used, since recently, in several domains. In this paper we examine potentials of using it in one novel domain — in automated evaluation of students programs. We propose an approach that provides precise assessment of functional correctness of student programs (while it does not address nor affect the teaching methodology). We describe our open-source, publicly available implementation of the approach, which is built on top of the compiler infrastructure LLVM and the software verification tool LAV. The results of evaluating the proposed approach on two real-world corpora of student programs and on a number of classic algorithms show that the proposed approach can be used as a precise and reliable supplementary technique in grading of student programs at introductory programming courses, algorithms courses and programming competitions.

Keywords: software verification, regression verification, automated evaluation of student programs, computer-supported education

1. Introduction

Despite many successful applications of software verification techniques, their potential is still to be explored in a number of new application domains. One domain are programming courses where automated evaluation of student programs is becoming progressively important. Namely, computer science is recognized as a fundamental field which is delivered in both universities and schools [69]. Also, the number of students enrolled at programming courses has rapidly grown over the last years [3]. Everyone benefits from automated evaluation [57,79]: the teachers get help in the grading process, while immediate feedback helps students in acquiring knowledge. The importance of automated evaluation is even more significant in the context of online learning where the adequate assessment is recognized as a challenging problem since contact with a teacher is minimal or even non-existent [61,73], while the number of students also grows quickly [56].

It is very important to provide a high quality, objective, precise and reliable automated evaluation [55]. Automated grading must (i) correctly classify correct and incorrect student solutions, (ii) correctly explain mistakes that students make, and (iii) run efficiently in practice [37]. There are different approaches for automated evaluation of student programs [2,35,58], considering many important aspects (e.g. functional correctness, code readability, modularity, complexity, efficiency). Various teachers and universities have various grading policies depending on such factors. In most cases, functional correctness is very highly valued [35] and in some educational settings is even essential. Such settings are commonly encountered at the university level at programming courses for future computer science majors and software engineers. Also, functional correctness is traditionally a must at IOI and ACM style programming competitions[1], which usually deal with problems that are very similar to problems taught at university level algorithms courses. Such algorithmic, competition style problems are also highly valued for employment in the high-end software companies and are usually asked at job interviews. In settings where students are required to produce fully functionally correct code and where subtle errors and hidden bugs are not allowed, attention must be put on all corner cases and it should be ensured that the grading process takes them into account.

Classifying correct and incorrect solutions of algorithmic problems is usually based on automated testing [17] and grading is performed solely by thorough testing on a number of test-cases. For example, this holds for online judges — web-platforms devoted to training for programming contests and interviews [23,31,50,51,65,72]. However, assessing functional correctness only by testing may give a misleading confidence since it may be error prone: the obtained results are directly influenced by the choice of test cases [78]. The problem is that the test cases are usually designed according to the expected solutions, while the teacher cannot predict all possible solutions and all important paths through a student solution. Moreover, no matter how well test cases are designed, testing cannot guarantee functional correctness [16]. Therefore, if a reliable automated evaluation is needed, it is necessary to apply some more involved techniques. A more promising choice are software verification techniques and we propose a verification based approach for improving classification of correct and incorrect solutions.

In this paper, we propose assessing functional correctness of students solutions by checking equivalence with teacher solutions. We are interested in showing equivalence of algorithmic problems that usually have short solutions, but can be very hard and complicated, thus their correctness can often be at stake. We describe how to apply formal static software verification techniques for assessing different kinds of equivalence of two programs and we focus on *regression verification* techniques. Development of regression verification techniques is often guided by applications in various industrial domains. The existing algorithms are advanced and there are still no general purpose implementations that are publicly available. Also, in the context of automated evaluation of programming assignments, it is necessary to adjust solutions in a way that makes these algorithms applicable, which also contains some nontrivial steps. Therefore, our work aims at enabling application of regression verification techniques in automated evaluation of programming assignments. We describe characteristics of programs that can be evaluated this way. We provide an open-source implementation of necessary transformations for automating this process. We present lessons learned from applying regression verification on three different corpora: a corpus of student solutions from an introductory programming course for computer science majors, a corpus of solutions submitted during national programming competitions, and a corpus of classic algorithms that are usually taught at algorithms courses. We show that, by our approach, functional correctness of significant amount of programs in introductory and algorithms courses can be automatically proved. We also show that our approach makes a good supplementary technique, aimed at the best solutions that successfully passed testing: it can reveal very subtle problems and point stu-

[1] IOI: `http://ioinformatics.org` , ICPC: `https://icpc.baylor.edu/`

dents to errors that they are not aware of. In the context of programming competitions, it can break ties and help differentiating the very best few competitors that qualify for next rounds. In some situations verification can even fully replace testing, eliminating the effort necessary to prepare tests.

Overview of the paper. Section 2 contains information about related work. Section 3 introduces our approach and describes its implementation. Section 4 gives results of experimental evaluation of the proposed approach with discussion of quantitative and qualitative analysis of capabilities of the approach. It also discusses possible threats to validity. In Section 5 we compare the proposed approach with other related approaches and tools. Section 6 gives conclusions and outlines possible directions for future work.

2. Related work

In this section we give a brief overview of related approaches and tools, both in the field of software verification and in automated evaluation of programming assignments.

Software verification and automated bug finding. Automated software verification tools aim to automatically check correctness properties of a given program or to find violations to some common features (the latter is known as automated bug-finding) [9]. There are different automated approaches [13,15,39] and there is a variety of tools based on these approaches like PEX [71], JPF [75], KLEE [10], CBMC [12], LAV [77]. CBMC and LAV are general purpose tools for statically verifying user-specified assertions and locating bugs such as buffer overflows, pointer errors and division by zero. CBMC is state of the art bounded model checker for C/C++ programs. LAV is primarily aimed at analysing programs written in the programming language C, but for the purpose of this work we have extended LAV with some constructs of C++ (used in the context of programming competitions, and present in our corpus).

Equivalence checking. Functional correctness of a program can be formulated in terms of precise formal specifications [32,43]. Also, it can be formulated in terms of the behavior of another program: two programs are equivalent if they exhibit the same behavior in all relevant aspects on all input values [26]. This includes checking termination and complexity of computation, but often only equivalence of outputs is considered [25]. The notion of correctness in this case has several positive aspects: it is not necessary to formulate a specification and, in general, checking equivalence of two programs is less computationally demanding than functional verification with respect to a formal specification [67]. Checking equivalence of two programs was considered already in 1960s [32], but the progress has been limited and not always practically applicable. Recent approaches introduced new possibilities [20,27]. There are different variations of program equivalence [25]. Programs are *partially equivalent* if any two terminating executions which start from equal inputs produce equal outputs. Another, weaker, notion of equivalence is *k-equivalence* — programs are k-equivalent if any two executions where loops and recursions have at most k iterations or calls, which start on equal inputs, produce equal outputs. The problem whether two programs are partially equivalent is an undecidable problem [68], while the problem whether two programs are k-equivalent (for some specific k, assuming that finite variable-domains are used) is decidable [25].

Regression verification. Applying testing to check whether two similar programs are equivalent is widely and intensively used in software development and is called regression testing [53]. *Regression verification* [20,67] attempts to achieve the same goals, but using techniques from formal verification. Here, checking equivalence means formally proving a mathematical statement about two programs that usually corresponds to some weaker form of equivalence. If successful, regression verification gives higher reliability since it guarantees full coverage [27]. Also, that it does not require additional expenses to develop and maintain a test suite. Since the problem of determining partial equivalence is undecidable [68], automating this process is challenging. Development of regression verification techniques is often guided by the application in different concrete areas, like security verification applications [4,62], multimedia systems [74], backward compatibility and refactoring [80], cryptographic algorithms [8,59], and hardware design [34]. General purpose automated regression verification techniques [6,20,27] are developed for large scale systems. These techniques consist of two steps: efficiently identifying functions that are affected by changes, and proving functional equivalence of these functions.

Functional correctness in automated evaluation. Automated testing is the most common way of evaluating student programs [17]. Test cases are usually supplied by the teacher and/or randomly generated [47]. Testing is used as an evaluation component of a number of web-based submission and evaluation systems [11,18,23,31,33,50,51,65,72]. Aside from checking functional correctness, testing can also be used for analysing efficiency, memory violations and run-time errors [1]. Software verification techniques are getting more commonly used in automated evaluation, usually for automated bug finding or for automated test case generation [36,37,71,78]. One formal approach for assessing functional correctness of student solutions, is based on rewriting techniques [40]. In this approach, it is necessary to write a formal specification of a desired solution.

Other important aspects in automated evaluation. There are other important aspects that are impossible or difficult to test or to be assessed by verification techniques, but that have to be taken into account in precise and high quality evaluation. For example, these are coding style, the design of the program, modularity, performance issues and the algorithm used. Therefore, other techniques are required for their assessment [29,52,54,70,78]. These techniques usually compare a predefined solution to the student solution. New approaches emphasize the importance of generating useful feedback for students [24,28,29,38,41,48,60]. Usually, the feedback is generated by failed test-cases or by peer-feedback [19,42,46]. Some approaches use both reference implementation and error model consisting of potential corrections to errors that students might make [64] and with this additional information are capable of making feedback that suggests possible corrections to incorrect student solution. Another kind of feedback is generated by computing behavioral similarity between two programs [45]. In this case, different metrics are used to calculate similarity to the model solution, which is then used as a measure of student progress. Machine learning techniques can be used for syntactically classifying similar solutions [55] or for clustering similar solutions by static and dynamic analysis [24]. The feedback is then generated by the teacher but only for each group of solutions.

3. Proposed approach and its implementation

In this section we discuss our open-source implementation based on regression verification techniques which is implemented on top of the software verification tool LAV [77], the LLVM system [44] and its C-language front-end Clang. We describe its implementation, as regression verification techniques are still rather new and advanced, and there are no implementations that are publicly available. Although regression verification is originally used for showing equivalence between two versions of the evolving program, we shall use it to show equivalence between the student and the teacher solution. The same techniques could be used for showing equivalence between different student solutions. The techniques described in this section and parts of our implementation can be adapted to work with other underlying verification systems by making an extension for specifying that some function calls should be encoded as uninterpreted functions calls.

Finding parts of code that are potentially equivalent is an important task for regression verification tools. There are different techniques for solving it (based on the analysis of control flow graphs and function names that preserve equivalent in different versions of programs [6,20,27]). In our setting, that problem is simple as corresponding functions are the teacher's and the student's solutions.

3.1. Regression verification in LAV

The input to the system LAV is a C program that may contain assertions, which can be accompanied by some assumptions (given width `assert`/`assume` function calls). Such assumptions are used to limit verification only to the cases allowed by the problem specification. User can put limits on the input variables in a way that subtle details get ignored or important preconditions are enforced (e.g., that some array is sorted). By enforcing additional assumptions, verification can be done against an arbitrary input specification.

In regression verification we try to prove the equivalence between the two solutions that are encoded by different functions that share the same interface. The implementation of these functions can be quite different (concerning used algorithms, computation that can be split into different auxiliary functions, etc.). Figure 1 contains different implementations of the function for finding maximum of three given numbers (these are all real-world examples, taken from our corpus described in Section 4.1, and reflect the possible diversity in solutions even for a very simple problem). To check equivalence of the functions `maxA` and `maxB` from Figure 1 using the system LAV, it is sufficient to verify the program illustrated in Figure 2. Calling the `assert` function in this program refers to the equality check of return values of these two functions (for arbitrary input values). Similarly, the function `maxC` can be shown to be equivalent to `maxA` and `maxB`. However, the function `maxD` contains a subtle bug and is not equivalent to the previous three ones. Since the C language does not allow returning arrays as function results, checking equivalence of functions that modify arrays is done by multiple assertions (Figure 2).

To verify an assertion, LAV encodes the asserted expression as a first-order logic formula and checks its validity by an underlying SMT solver [7]. We will focus on integer variables that are modelled either by the theory of linear arithmetic (LA), or by the theory of bit-vector arithmetic (BVA). Although there are important semantic differences between LA and BVA, in the context of education some of these differences are not relevant (for example, at introductory level, overflows/underflows are usually not considered). LA is very efficient, but does not support many operators that BVA supports and that are used in C-programs. For efficiency reason, BVA will be used only when that is necessary. We will focus on programs containing loops and/or recursive functions, since their treatment is the most delicate aspect in verification. Since loops are not supported in SMT formulas, functions have to be transformed into some loop-free form. We will consider

```
int maxA(int x, int y, int z) {     int maxB(int a, int b, int c) {
   int m = x;                         int max;
   if(y > m) m = y;                   max = a;
   if(z > m) m = z;                   if (b>max && b>c) max=b;
   return m;                          else if(c>max) max=c;
}                                     return max;
                                    }

int maxC(int i, int j, int k) {     int maxD(int o, int p, int q) {
   int max;                           if(o>p && o>q)
   if(i>j && i>k) max= i;                 return o;
   else if(j>k) max = j;              else if(p>o && p>q)
   else max = k;                          return p;
   return max;                        else
}                                         return q;
                                    }
```

Fig. 1. Different implementations for determining the maximum value

```
#include "maxAB.h"
int main() {
  int a, b, c;
  scanf("%d%d%d",
        &a, &b, &c);
  assert(maxA(a,b,c) ==
         maxB(a,b,c));
  return 0;
}
```

```
#include "modifyAB.h"
int main() {
  int i; char s[MAX], t[MAX];
  scanf("%s",s);
  for(i = 0; s[i]; i++)
    assume(t[i] == s[i]);
  resultA = modifyA(s); resultB = modifyB(t);
  assert(resultA == resultB);
  for(i = 0; s[i]; i++)
    assert(t[i] == s[i]);
  return 0;
}
```

Fig. 2. Checking equivalence of two functions: (left-hand side) functions from the Figure 1 and (right-hand side) functions that modify contents of arrays

two different techniques for loop elimination: (i) loop unrolling for proving k-equivalence (ii) transforming loops into recursive functions and then using uninterpreted functions to express the inductive hypothesis [27,67].

K-equivalence by loop unrolling. Functions that contain loops with a fixed upper bound can be transformed into equivalent functions that do not contain loops. However, unrolling loops a large number of times may introduce complex formulas that cannot always be efficiently reasoned about. Checking equivalence of functions with arbitrary loops is a major challenge and is generally not solvable. Therefore, we must resort to using some approximation. For example, instead of proving equivalence of two functions we can try proving their k-equivalence. In such case, loops are unrolled k times, for some given value k. Figure 3 shows a loop that is unrolled $k = 3$ times. When proving k-equivalence, the choice of an appropriate value for k is very important. Higher values of k are giving a higher level of confidence to the code under evaluation, but increasing k can introduce scalability issues. On the other hand, some verification tools rely on common experience that many errors can be discovered in only one loop iteration [5,21]. Note that the number k often corresponds to the length of the input series for which the algorithm is verified, although this need not be the case always (for example, in binary search, unrolling loop for k times guarantees the correctness for the arrays with at most 2^k elements). In our experiments, we usually used $k = 5$, as for this value the analysis was efficient and results showed to be reliable. We discuss this choice in more details in Section 4.1. Similar to loop unrolling is the recursive function call unrolling. However, recursive function unrolling

```
float mean_valueA(int a[], int n) {
  float s = 0;
  int i;
  for (i=0; i<n; i++)
    s += a[i];
  return s/n;
}

float mean_valueB(int a[], int n) {
  int i;
  float m;
  m = i = 0;
  while(i < n)
    m = m + a[i++];
  m = m/n;
  return m;
}
```

```
float mean_valueA_k3(int a[], int n) {
  float s = 0;  int i;
  i = 0;
  if(i < n) {
    s += a[i];
    i++;
    if(i < n) {
      s += a[i];
      i++;
      if(i < n) {
        s += a[i];
        i++;
      }
    }
  }
  return s/n;
}
```

Fig. 3. Calculating the mean value of an array (left-hand side), unrolling $k = 3$ times a loop of the function `mean_valueA` (right-hand side)

can lead to significantly slower verification, due to introduced stack-frame modeling, and due to exponential code growth when there is more than one recursive call.

Partial equivalence by uninterpreted functions. Instead of loop unrolling, in some situations we can use inductive reasoning to prove partial equivalence between the two functions by using uninterpreted functions to model inductive hypothesis [27]. To succeed in proving partial equivalence by uninterpreted functions in programs that contain loops it is necessary to have solutions where entry point, exit condition, and loop invariant are the same (while the body of the loop can differ).

Preprocessing. There are several constructs in C that complicate elimination of loops (e.g., `break`, `continue` and `return`), and in the preprocessing phase we automatically transform the program to eliminate such constructs. Also, we transform all loops to the `while` loop. Removing `return` statements is illustrated in Figure 4. If the `return` statement occurs within a nested loop, the transformation is applied once for each loop, starting from the innermost loop. This transformation introduces a special value `RET_UNDEF` that cannot occur as the return value of the function. Similar transformations are applied to eliminate `break` and `continue` statements.

```
while(<cond>) {
   ....
      <return> <val>;
   ....
}
```

```
<retvar> = RET_UNDEF;
while (<cond> && <retvar> == RET_UNDEF) { ...
      <retvar> = <val>;
   if (<retvar> == RET_UNDEF)
...}
if (<retvar> != RET_UNDEF)
   return <retvar>;
```

Fig. 4. Preprocessing transformations: `return` statement elimination

Introducing uninterpreted functions. Consider the functions given on top of Figure 5 (also taken from our corpus). After preprocessing the next step is to transform loops into recursive functions (as illustrated in the middle of Figure 5). An important requirement (that is often satisfied) is that the loop changes exactly one variable that is alive after the loop (its value is read and used before it is eventually changed). In the function idx_minA, the variable min is such a variable and in the function idx_minB, the variable idx is such a variable. Then, the recursive equivalent of the loop will be a function whose return value will be exactly that variable. The function can have many input parameters (the variables that are accessed within the loop, except the ones that are declared in the loop or are always assigned a value before their value is read). Equivalence of the recursive functions can be proved by induction on the number of recursive calls made during their execution. The base case is when no recursive calls are made. As the induction hypothesis we can assume that the statement will hold for recursive calls i.e., that recursive calls return the same values. Under that assumption and the definition of the recursive functions it should be proved that the statement holds i.e., that the functions return the same values in the case when recursive calls are made (in the code on Figure 5, that is when i < n). The crucial part of the technique is to encode such induction hypothesis by replacing recursive calls by a call to an uninterpreted function (as illustrated at the bottom of Figure 5). After those replacements, we are left with a loop-free and recursion-free functions that can be shown equivalent using the techniques for loop-free, recursion-free programs. Once the recursion is removed, there is no need to have auxiliary functions representing loops, thus, for simplicity, they can be inlined back (as illustrated on the bottom of Figure 5). A more complicated example from our corpus is given in Figure 6. An important question is how to order parameters of uninterpreted functions (since solutions must use the same order of parameters). The names of the variables, and the order of their declarations can vary between alternative solutions, therefore some kind of semantic matching between the corresponding variables is needed. Currently, to solve this problem, our transformation uses a heuristic: parameters are first ordered by their type, and then by their name. In most cases students use canonical variable names (e.g., min for a minimum value), and the heuristic works. However, when it fails, all possible combinations of parameter ordering can be checked (usually there are not many parameters).

3.2. Interpreting results of regression verification

When using regression verification in evaluation process, it is crucial to correctly interpret obtained results and to understand relationship between different evaluation techniques.

Function calls. Inlining is the only fully precise technique for modeling function calls. Other techniques incur loss of information about the exact program behavior. Therefore, when other techniques are used, it might not be possible to prove the equivalence.

K-equivalence vs partial equivalence. The fact that functions are k-equivalent for some value k, does not guarantee that these functions are partially equivalent, or even k-equivalent for some larger value k. It only guarantees that these functions will give same outputs for each input value, if restricted to k or less loop iterations. However, in our experimental evaluation in both our corpora, we did not find two functions that were k-equivalent and not partially equivalent (with exception to the functions that used unmodelled library function calls, which were detected independently). This is partially due to the fact that these programs were also thoroughly tested and checked for bugs before checked for k-equivalence. However, if two functions are not k-equivalent for some value k, then that means that these functions are not equivalent. In our corpora, there were several cases where k-equivalence discovered a bug that the testing missed (Section 4.1).

K-equivalence vs testing. Like testing, k-equivalence can always be applied. Proving k-equivalence is usually much stronger information than information obtained by testing.

```
int idx_minA(float a[], int n) {
  int min = 0; int i;
  for (i = 1; i < n; i++)

    if (a[i] <= a[min])
       min = i;
  return min;
}
```

```
int idx_minB(float a[], int n) {
  int i, idx; float min;
  for (i = 1, min = a[0], idx=0;
       i < n; i++)
     if (min >= a[i]) {
       min = a[i];
       idx = i;
     }
  return idx;
}
```

```
float idx_minA(float a[], int n) {
  int min = 0; int i = 1;
  min = idx_minA_loop(a, n, i, min);
  return min;
}

float idx_minA_loop(float a[],
                    int n,
                    int i,
                    int min) {
  if (i < n) {
    if (a[i] <= a[min])
       min = i;
    i++;
    min = idx_minA_loop(a, n, i, min);
  }
  return min;
}
```

```
float idx_minB(float a[], int n) {
  int i, idx; float min;
  i = 1, min = a[0], idx=0;
  idx = idx_minB_loop(a, n, i, min, idx);
  return idx;
}

int idx_minB_loop(float a[], int n,
                              int i, int min,
                              int idx) {
  if (i < n) {
    if (min >= a[i]) {
       min = a[i];
       idx = i;
    }
    i++;
    idx = idx_minB_loop(a, n, i,
                             min, idx);
  }
  return idx;
}
```

```
float idx_minA(float a[], int n) {
  int min = 0; int i = 1;
  if (i < n) {
    if (a[i] <= a[min])
       min = i;
    i++;
    min = uf(a, n, i, a[min], min);
  }
  return min;
}
```

```
float idx_minB(float a[], int n) {
  int i, idx; float min;
  i = 1, min = a[0], idx=0;
  if (i < n) {
    if (min >= a[i]) {
       min = a[i];
       idx = i;
    }
    i++;
    idx = uf(a, n, i, min, idx);
  }
  return idx;
}
```

Fig. 5. Finding the index of the minimum element: implementation, transformation into recursive function and replacement by an uninterpreted function

By testing, it is checked that for one single set of inputs the functions give the same outputs. Here we are not restricted to the possible inputs, but just for the number of loop iterations. For example, if we prove that functions shown in Figure 3 are k-equivalent for $k = 5$, that means that for each array of the size 5 or less, these functions calculate the same outputs. This is equivalent to testing the functions with $\Sigma_{i=1}^{5}(2^{(sizeof(int))})^{i}$ different test cases, which, for $sizeof(int) = 32$, approximately equals to 1.46×10^{48}. Also, complete path coverage is achieved in all cases were loop iterations are restricted to k. However, there are situations when checking k-equivalence does not provide better information compared to testing, like if there is only one input value and the number of loop iterations together with the resulting values are controlled only by this value.

```
int strcspnA(char s[], char t[]) {
  int i, j;
  for(i=0; s[i]; i++) {
    for(j=0; t[j]; j++)
      if(s[i] == t[j])
         return i;
  }
  return -1;
}
```

```
int strcspnA (char s[], char t[]) {
  int i, j;
  i = 0; int ret1 = RET_UNDEF;
  if (s[i] && ret1 == RET_UNDEF) {
    j = 0; int ret2 = RET_UNDEF;
    if (t[j] && ret2 == RET_UNDEF) {
      if (s[i] == t[j]) ret2 = i;
      if (ret2 == RET_UNDEF) j++;
      ret2 = uf1(ret2, i, j, s, t);
    }
    if (ret2 != RET_UNDEF) ret1 = ret2;
    if (ret1 == RET_UNDEF) i++;
    ret1 = uf2(ret1, i, s, t);
  }
  if (ret1 != RET_UNDEF) return ret1;
  return -1;
}
```

Fig. 6. Transforming a function with double `for` loop and a `return` inside

Partial equivalence vs uninterpreted functions. For proving partial equivalence by uninterpreted functions, it is necessary to perform the described transformation. If equivalence of two transformed functions is proved, then the original functions are also equivalent. Both the entry point to a loop and the loop-exit condition influence the proof of this equivalence [27]. Therefore, it can be useful to have several model solutions. If equivalence of the two functions cannot be proved, then that does not imply that the original functions are not equivalent: it may happen that their equivalence only cannot be proved this way. There is a number of such examples [27], but, in practice, there are many cases where this technique can be successfully applied (discussed in Section 4.2).

Uninterpreted functions vs k-equivalence. The obvious advantage of using uninterpreted functions to k-equivalence is that partial equivalence is a stronger property. Also, using uninterpreted functions is usually more efficient than loop unrolling with a high value for k. However, uninterpreted functions model only changes captured by a single scalar return value. Therefore, they cannot be applied when more than one variable is modified in a loop that is live after the loop, or when the loop modifies values of an array.

4. Evaluation and results

To illustrate applicability of regression verification, we analyzed two corpora of problems solved by students and a corpus of classic algorithms that are usually taught at introductory and algorithms courses. Regression verification, in the first context, refers to determining equivalence of solutions provided by the teacher and by the student, and in the second between all pairs of different proposed solutions. All experiments were performed on a computer with an Intel Core i3-4000M on 2.40GHz and with 3.9GB of RAM.

4.1. Verifying student solutions

We have conducted an experimental evaluation on two real-world corpora: one from an university introductory programming course for computer science majors (we call this corpus *exam corpus*), and the other from the national programming competitions (we call this corpus *competition corpus*). We chose to use these corpora as automated evaluation is especially important when the number of enrolled students is large, and these are good examples of such situations. In both corpora we analyzed the programs that: (i) successfully compile; (ii) pass all manually designed test cases (12 per problem for the first, and between 15 and 25 for the second corpus);[2] (iii) where a bug-finding tool (we used LAV) does not find any bugs. We chose to use such programs since they are expected to be functionally correct: programs that fail to meet the above requirements are obviously not functionally equivalent to the model solutions, thus there is no need to further analyze them. Also, another important reason for using these corpora is that testing (sometimes enhanced by automated bug finding) is an established approach widely used for automated evaluation. Therefore, by using corpora that successfully pass manually designed test cases and where a bug-finding tool does not find any bugs, we wanted to demonstrate that the proposed approach can add value to preciseness of the automated evaluation.

Both corpora, problem descriptions and the used test cases are publicly available [76]. Statistics showing the number of problems, distribution of number of solutions per problems, lines of code and cyclomatic complexity [49] are given in Table 1.

Table 1. Distribution of number of solutions per problem, lines of code (LOC) and cyclomatic complexity (CC) per solution

Exam corpus 12 problems, 224 solutions, 4104 LOC						**Competition corpus 10 problems, 214 solutions, 4857 LOC**						**Classic algorithms 59 problems, 159 solutions, 2007 LOC**					
	Min	Max	Avg.	Med.	Std. dev.		Min	Max	Avg.	Med.	Std. dev.		Min	Max	Avg.	Med.	Std. dev.
Solutions per prob.	3	44	18.67	18	12.84	Solutions per prob.	4	39	21.4	22.5	12.26	Solutions per prob.	2	7	2.69	2	1.16
LOC per sol.	5	62	18.32	19	11.54	LOC per sol.	5	83	22.7	21	11.97	LOC per sol.	3	36	12.78	11	6.95
CC per sol.	1	17	6.22	6	4.01	CC per sol.	1	50	9.01	8	6.86	CC per sol.	2	28	5.85	4	4.07

A) Description of the corpora

Exam corpus consists of programs written by students, during programming exams in the programming language C [78]. Originally 1277 solutions to 15 given problems were collected. Programs that do not compile or do not pass testing (1011 solutions), and programs that contain memory violations or other bugs (additional 35 solutions) were eliminated. Three problems (28 solutions) were not suitable for regression verification (in two cases it was more efficient to thoroughly test these solutions, as described in Section 3.2, while in one case the problem requires complex data structures not supported by our verification tool). The filtered corpus consists of 12 problems (203 solutions).

Competition corpus consists of programs written by primary school pupils (aged 12 to 16) competing at the national competitions in Serbia (in 2017).[3] This competition is organized in accordance to International Olympiad in Informatics (IOI) guidelines, and scoring and ranking is done solely based on results obtained by automated testing on test-cases, prepared in advance. Among four stages, we considered the second and the third

[2] Test cases were carefully designed for grading purposes and contain different important usage scenarios. For all teacher solutions, 100% of code coverage is achieved by these tests, measured by *gcov* tool [22].

[3] The competition is organised by Mathematical Society of Serbia which is a member of European Mathematical Society. The site of the competition is `https://dms.rs/informatika-osnovne-skole/`

stage. For the first stage, there was no central repository of solutions, while for the forth stage there were just a few solutions that passed testing. We considered 10 different problems with 629 solutions written in C/C++ (that was around 80% of all submitted solutions, other solutions were written in Pascal, Small Basic and C#). After the programs that do not compile or pass testing (411 solutions) and the programs where the bug finding tool detected bugs (4 solutions) were eliminated, the final corpus consists of 214 programs.

Differences between these two corpora. In the exam corpus, student solutions are required to be robust and report errors for incorrect inputs, while in the competition corpus it was allowed to assume that the input is always correct (in accordance to the problem specification). Also, after the testing-phase, student solutions were manually inspected and modularity, readability and other aspects were additionally graded. On the other hand, structure and modularity of programs written during competition was quite bad (usually everything was contained in the `main` function), which made them harder to verify.

Preparing for verification. To aid verification, the teacher and the student solution should be represented as separate functions that take their input and return output solely through function parameters and the return value. However, in most cases the student solution was implemented within a function that reads the data from the standard input and writes the results on the standard output (especially in the competition corpus). Therefore, our implementation supports an automatic transformation that does the function extraction. The transformed programs are also publicly available [76].

The functions that read the input data and contain assertions that teacher and student solution match (like in the programs from Figure 2) were manually written. In the exam corpus, these functions were simple, including only necessary assertions, while in the competition corpus, these functions contained all necessary additional constraints on input values (imposed by problem descriptions).

B) Results

The results are summarized in Table 2. The problems from both corpora can be divided into two types. The first type of problems (denoted as A) does not require using loops in their solutions or only requires the use of bounded loops. For this type of programs, the used approach is exact, i.e. it does not make neither false positives nor false negatives. The second type of problems (denoted as B) requires the use of loops in their solutions. These do not have upper bounds or have high upper bounds that cannot be completely unrolled due to time and memory limits.

Table 2. The results of regression verification applied on exam and competition corpus

Type of problem	Corpus	Num. of problems	Num. of solutions (total / per problem)	Num. of functions	Equivalent by RV/manually	Non-equivalent by RV/manually
(A) Problem requires using bounded loops or no loops	Exam corpus	6	136 / [3,7,18,31,33,44]	136	129 / 129	7 / 7
	Competition corpus	5	88 / [4,4,12,29,39]	88	77 / 80	8 / 8
	Total	11	224	224	206 / 209	15 / 15
(B) Problem requires using high upper bounds or no bounds	Exam corpus					
	– UF + k-equivalence	2	41 / [20,21]	62	38 + 16 / 54	8 / 8
	– only k-equivalence	4	26 / [4,5,7,10]	26	20 / 20	6 / 6
	Competition corpus					
	– only k-equivalence	5	126 / [16,20,25,32,33]	126	106 / 111	15 / 15
	Total	11	193	214	180 / 185	29 / 29

Exam corpus. The exam corpus contains 6 problems with 136 solutions of type A. LAV successfully shows equivalence for 129 (correct) solutions and finds 7 solutions that are not functionally equivalent to the model solutions. The exam corpus contains 6 different problems with 67 solutions of type B.

Partial equivalence using uninterpreted functions: For two problems with 41 solutions and 62 pairs of checked functions, it was possible to check equivalence by uninterpreted functions and in 38 cases the equivalence is proved. For remaining 24 functions, equivalence cannot be proved by uninterpreted functions. In these cases, we checked for k-equivalence for $k = 5$, and 16 functions are proved k-equivalent, while 8 functions (in five different solutions) are proved to be non k-equivalent.

k-equivalence by loop unrolling: The remaining 4 problems with 26 solutions cannot be modelled by uninterpreted functions. We tried proving k-equivalence, and decided to use $k = 5$ as we find it a reasonable compromise between scalability and reliability of obtained results (scalability is discussed in Section 4.2). In many cases, $k = 5$ corresponds to equivalence checking of an algorithm that is applied on all arrays of the maximum size 5 and obtains full path coverage in such cases. Knowing the nature of these problems, we expected that there should not be a significant difference between, for example, an array of the size 5 and an array of a bigger size, and our experimental results confirmed this assumption, showing that even smaller values for k could have been used without compromising preciseness of the results. LAV successfully proved k-equivalence of 20 solutions which are indeed functionally equivalent (based on manual check). For the remaining 6 solutions, LAV proved non k-equivalence. All non k-equivalent solutions could have been found already with $k = 2$.

The time for program transformation was negligible. The average time for verification per solution was 0.7s, while the median value was 0.05s. The LA theory was used whenever it was possible, as it provides faster verification. Otherwise, BVA was used. For example, LAV generates formulas and verifies functional equivalence of functions `maxA` and `maxB` (from Figure 1) with respect to the theory of LA and the Z3 SMT solver in 0.02 seconds. If a solver for the BVA theory is used, then the time necessary for proving this equivalence is 0.16 seconds with the SMT solver Boolector, and 0.84 seconds with the SMT solver Z3. In an analogous way, LAV can also prove that the functions `maxA` and `maxD` are not functionally equivalent. The time needed for this is 0.02 seconds in the context of linear arithmetic, and 0.09 in the context of the theory of bit-vectors. LAV generates a counterexample ($a = 29$, $b = 29$, $c = 30$), i.e. the values of the variables for which this equivalence does not hold. This counterexample can be useful for understanding the bug in the function `maxD`. Proving partial equivalence by using uninterpreted functions was usually faster than showing k-equivalence. Proving partial equivalence of functions from Figure 5 takes 0.028 seconds in the context of LA, and proving 5-equivalence of functions from Figure 3 takes 0.068 seconds.

Competition corpus. Results for the competition corpus are very similar. Because of the poor modularity and almost total absence of user defined functions in the code, on this corpus we could not apply regression verification with uninterpreted functions. In order to check the claim that similar verification tools can also be used for this purpose, in addition to LAV, we ran the CBMC tool. As expected, we got the same results. There were 8 solutions in this corpus that contain library function calls that are not precisely modelled by both tools or that contain advanced C++ concepts (from standard template library) that are unsupported by both tools. Therefore, these solutions were not considered. The results are summarized in Table 2. The average time per solution for CBMC was 1.4s, while for LAV it was 7s. The median value for CBMC was 0.8s and for LAV was 0.7s. We also applied random testing to all programs in this corpus (we generated fresh 25 random tests for each task), but random testing detected bugs in only 2 solutions.

C) Discussion

We performed a quantitative analysis of the obtained results to assess in what extent the proposed approach can add to quality of automated evaluation. We also performed a detailed qualitative analysis of the obtained results to detect in what situations it can be expected to get the most from the proposed approach.

Quantitative analysis of results. The percentage of non-equivalent solutions is approximately the same in both corpora: it is around 10% of all solutions that have been analyzed (and graded as being functionally correct). It is relatively low since programs were thoroughly tested and bug finding tool was applied.[4] However, it is definitely not negligible and reveals that in spite of very thorough testing and bug-finding, around 10% of programs still contain bugs that go undetected. This illustrates the limited power of these approaches and shows that the proposed approach can add to the quality and precision of automated evaluation.

Qualitative analysis of results. We manually analyzed all programs that were shown to be non-equivalent to the model solution, in order to detect what kind of bugs are found by regression verification. In the following text we summarize such examples.

Completely different logic valid in most cases. There were some solutions that are very different from the expected solution and which work for most input values. For example, one model solution required calculating $\lceil\frac{x}{3}\rceil \cdot \lceil\frac{y}{3}\rceil$ while a student submitted a solution with 47 lines of code that introduced 9 auxiliary variables, with 10 different branches. Another example (found in several solutions) is comparison of two dates by converting them to integers, using the formula $d + m \cdot 30 + y \cdot 365$.

Missing branches. In several cases, students introduced unnecessary branching which left the input uncovered. One such example is illustrated on Figure 7 where the branch for $K < 5$ and $R = 5$ is missing. It is hard to cover such situations by test-cases, since the branching is not the part of the problem semantics, but is artificially introduced by the student. In some cases, branching ends with an `else` branch and all missing branches will execute its code. For example, the function `maxD` in Figure 1 contains such an error, i.e. the branch for $o = p$ and $q < o$ is missing.

Specific input series. Some errors were due to wrong behavior of programs when the input series of numbers were specific in some sense, for example, series containing just a single element or series containing elements in some specific order. Although such errors could be caught by careful testing, it is hard to predict all such special inputs in advance. Applying regression verification removes the burden from the test designer, making grading much more reliable.

[4] Automated bug-finding in this context searched for bugs such as buffer overflows, division by zero or null pointer dereferencing. For the exam corpus, a detailed testing and automated bug finding is described in [78]. The same approach is applied in the case of the competition corpus.

```
if(K<5 && R<5)
  i=(K/2)*(R/2);
else if(K<5 && R>5)
  i=(K/2)*(((R-1)/3)+1);
else if(K>=5 && R<5)
  i=(((K-1)/3)+1)*(R/2);
else if(K>=5 && R>=5)
  i=(((K-1)/3)+1)*(((R-1)/3)+1);
```

```
int y;
while (yr > 0) {
  y = y + 1;
  yr = yr - 3;
}
```

Fig. 7. A missing branch (left-hand side) and uninitialized variable (right-hand side)

Uninitialized variables. In several cases uninitialized local variables were used, like the solution from the competition corpus which contained the code shown on Figure 7. Although the initial value of local variables at run-time cannot be predicted, in many cases it is zero (and it is usually the correct initial value) and the tests pass.

Potential errors in variable range. In some cases, solutions used constructs that could potentially introduce integer overflows. In the concrete tasks, limits were such that those solutions were detected to be safe. However, if the assumptions for the limits are removed, the verification detects non-equivalence and this could be used to signal potential errors to novice programmers. For example, for sorting three integer variables some solutions found the minimum, the maximum, and calculated the middle one as the sum minus the minimum and the maximum. A similar situation was comparing dates by converting them to integers using the formula $1000 \cdot y + 50 \cdot m + d$ or when maximum/minimum is initialized to arbitrary values (in our corpus, we have seen minimum being initialized to 1000000000, 454545454, 1000, 9990000, 12345, and 99999999). Regression verification detects these solutions as non-equivalent when no additional assumptions are given, and that can be used to warn programmers about bad programming style and potential errors.

Most of the errors were found in programs containing rich branching structure. Programs that do not contain branching (whether or not they contain loops) and that pass testing, usually do not contain errors or contain only errors detected by bug finding (buffer-overflows, division by zero etc.). On the other hand, programs where control flow can follow various paths are much harder to verify only by testing and applying regression verification is most beneficial in such situations. A good indicator where regression verification can be beneficial is the presence of solutions that fail just in a few test cases. That indicates that some solutions failed only in some branches, and it is reasonable to expect that the solutions that passed all the tests could also contain errors (in some other branches that were not covered by test cases).

4.2. Verifying classic algorithms

To illustrate the type of problems that can be assessed by regression verification, we have applied regression verification to same standard algorithms that are usually covered in introductory and algorithms courses [14,63]. Detailed problem descriptions and corresponding source codes are available on web page [76] together with the two other corpora. Statistics summarizing the number of problems and solutions, their length, and cyclomatic complexity are given in Table 1.

A) Description of the corpus

We applied our approach on some loop free algorithms (most of them based on different forms of branching, like branching based on discrete values, intervals, lexicographic comparison, or hierarchical nested branching) and on some programs with loops, using the technique of uninterpreted functions (algorithms that calculate statistics, perform linear search and filter series, map all series elements by applying a given transformation, and various combinations of such algorithms). K-equivalence can be successfully applied on a wide set of problems. We examined 15 problems with 59 fundamentally different solutions that yielded 100 pairs that were checked for k-equivalence. For example, we have considered the problem of finding a sub-array of contiguous elements with the maximal sum and have shown k-equivalence between the brute-force solution, its optimized variant based on two-pointer technique, the solution based on Kadane's (dynamic programming) algorithm, the solution based on maximizing the difference between the array partial sums, and a solution based on the recursively implemented divide-and-conquer approach.

B) Results

Times needed for showing partial equivalence are summarized in Table 3, showing that if this approach is applicable, then the verification is usually very fast. Distribution of times needed for showing k-equivalence for different values of k in more advanced algorithms using CBMC are summarized in Table 4. Table shows that required verification times quickly grow. Verifying recursive solutions is the most time consuming: all 10 cases where the timeout of 60 seconds was violated for $k = 5$ involved at least one recursive solution. We also applied LAV on 32 non-recursive solutions (since it does not support recursive function unrolling) and the results were very similar.

Table 3. Partial equivalence of classic algorithms: number of problems, solutions and checked pairs of solutions; minimum, maximum and median time in seconds

Group name	Num. of problems	Num. of solutions	Num. of pairs	Min.	Max.	Median
Loop free programs	14	36	33	0.01	0.27	0.01
Loops – unininterpreted functions	30	64	35	0.01	2.68	0.01

C) Discussion

Proving functional equivalence of two equivalent solutions is more time demanding than finding a difference between two non-equivalent solutions. The reason is that in proving functional equivalence all possible paths through two different solutions must be analyzed, while for finding a difference all possible paths through solutions are analyzed only in the worst case. Since our analysis applied on classic algorithms corpus included only functionally equivalent solutions, this suggests that the same or less amount of time is needed in case of considering non-equivalent solutions of the proposed problems, which is an important use-case in context of students solutions.

Table 4. k-equivalence of classic algorithms for different values of k, where each pair of solutions is checked for equivalence — first row: a number of proved pairs (time out set to 60 seconds) vs. number of all pairs; second row: min.–max.(median) times (in seconds)

Problem name (Num. of solutions)	**proved/all pairs min - max (median)**			
	$k = 3$	$k = 4$	$k = 5$	$k = 6$
1. Search (5)	10/10	10/10	9/10	7/10
	0.17 - 9.6 (0.42)	0.23 - 13.79 (2.94)	0.4 - 43.31 (1.86)	0.59 - 21.49 (4.68)
2. Sort (7)	21/21	21/21	13/21	6/21
	0.18 - 3.35 (1.44)	0.44 - 22.93 (6.29)	1.56 - 58.07 (38.86)	11.16 - 21.89 (17.57)
3. K-th element (3)	3/3	3/3	2/3	0/3
	0.35 - 3.14 (1.03)	1.45 - 55.04 (4.96)	12.34 - 25.87 (19.105)	-
4. Majority (3)	3/3	3/3	3/3	3/3
	0.12 - 0.31 (0.3)	0.15 - 0.63 (0.63)	0.18 - 1.17 (1.12)	0.26 - 3.82 (2.95)
5. Fibonacci (4)	6/6	6/6	6/6	6/6
	0.09 - 0.36 (0.22)	0.1 - 0.72 (0.4)	0.1 - 1.58 (0.78)	0.1 - 3.82 (1.79)
6. Longest increasing subsequence (4)	6/6	6/6	6/6	6/6
	0.14 - 0.22 (0.18)	0.16 - 0.46 (0.34)	0.3 - 1.31 (0.69)	0.53 - 6.32 (2.48)
7. Min. coins (6)	15/15	15/15	15/15	15/15
	0.28 - 0.29 (0.28)	0.41 - 0.43 (0.42)	0.9 - 0.93 (0.92)	5.84 - 5.93 (5.86)
8. Stock span (2)	1/1	1/1	1/1	1/1
	0.15 - 0.15 (0.15)	0.34 - 0.34 (0.34)	1.14 - 1.14 (1.14)	5.63 - 5.63 (5.63)
9. Max. segment - sum (5)	10/10	10/10	10/10	5/10
	0.12 - 0.56 (0.18)	0.3 - 2.35 (0.56)	0.83 - 24.85 (5.48)	2.79 - 29.56 (20.2)
10. Num. of segments (3)	3/3	3/3	3/3	0/3
	0.21 - 0.25 (0.22)	1.36 - 2.83 (1.66)	13.75 - 56.06 (22.75)	-
11. Num. of pairs - sum (3)	3/3	3/3	3/3	3/3
	0.15 - 0.33 (0.24)	0.25 - 0.59 (0.49)	0.66 - 1.69 (1.23)	1.44 - 4.29 (3.78)
12. Num. of pairs - diff (3)	3/3	3/3	3/3	0/3
	0.28 - 0.70 (0.43)	1.78 - 6.45 (3.05)	14.87 - 45.67 (39.9)	-
13. MaxPairing (5)	10/10	10/10	10/10	6/10
	0.18 - 0.31 (0.26)	0.32 - 1.01 (0.61)	0.66 - 7.99 (1.52)	1.64 - 4.91 (3.19)
14. Longest palindromic substring (3)	3/3	3/3	3/3	3/3
	0.17 - 0.55 (0.41)	0.26 - 1.96 (1.52)	0.4 - 6.84 (5.10)	0.6 - 19.89 (15.43)
15. Prefix - suffix (3)	3/3	3/3	3/3	3/3
	0.13 - 0.14 (0.13)	0.16 - 0.20 (0.18)	0.28 - 0.34 (0.28)	0.41 - 0.55 (0.43)
Total (59)	100/100	100/100	90/100	64/100
	0.09 - 9.6 (0.28)	0.1 - 55.04 (0.53)	0.1 - 58.07 (1.46)	0.1 - 29.56 (5.27)

4.3. Threats to validity

Our experimental results give good promises for real-world applications in education, but their generalization to other situations have to be discussed.

Languages C/C++ are present at introductory programming and algorithms courses at many leading universities, but are not the most popular choices [30]. Although the tools we use are tailored for C/C++, the proposed approach can be adapted for other languages.

We do not consider equivalence of bigger sized projects, but this issue could be addressed by showing equivalence of their smaller parts (like modules or functions). The problem of detecting and aligning parts of code is discussed in regression verification [6,20,27], but that is an orthogonal problem to the one we studied. Also, different solutions of students projects might be completely different, as such projects are usually not given by strict specifications, while a certain level of creativity is even expected.

Finally, there is a question of the number of errors that are undetected by testing and bug finding and are found by regression verification. In our experiments, percentage of such programs was around 10% in both corpora. Obviously, the percentage depends on how test-cases are designed, but since our corpora are taken from real-world exams and competitions, we expect that these are representative or at least very illustrative examples. By the detailed analysis of the detected errors, we have shown that there are situations where test cases are very hard or almost impossible to predict in advance. Therefore, applying the proposed approach releases the burden of creating such tests.

5. Relationship to other approaches

In Section 2 we identified all important aspects of the related work, and here we discuss them in the context of the proposed approach.

Regression verification techniques customized for specific domains [4,8,34,59,62,74,80] are usually only semi-automated and require additional information from an expert (like inductive invariants). However, it is not likely that students, or even teachers, would be able to provide such expertise. Therefore, these customized techniques do not seem applicable for the evaluation of student solutions. In regression verification of large systems [6,20,27], program behavior is usually checked only for k-equivalence [6], while we consider both k-equivalence and partial equivalence in order to get more accurate results. In regression verification of student programs, the code is usually short and, in contrast to regression verification of large systems, it is not difficult to determine which functions should be checked for equivalence. On the other hand, checking equivalence of matched functions in our context can be very challenging since these functions did not evolve from the same code and therefore may have a high level of diversity.

The rewriting-based approach [40] for verification of student programs requires formal specifications to be available. An evaluation of this approach was performed on a corpus consisting of 41 programs (all chosen to be functionally correct) – solutions of 5 different simple problems. The student solutions were transformed manually and the system successfully verified 27 programs out of 41. The downside of this approach is that it can be difficult task for a teacher to create formal specifications. In our approach, code transformations are performed automatically. The corpora used in our evaluation are bigger and wider, hence lead to more conclusive results.

Concerning evaluation based on automated testing [11,18,23,31,33,50,51,65,72] and automated bug-finding [36,37,71,78], our approach is complementary and it gives an additional confidence on functional correctness of the program. We think that the best way to use it is to apply it on programs where cheaper techniques such as testing and automated bug-finding did not discover any bugs. Still, it can also be used complementary to the grading techniques which do not asses functional correctness (e.g. grading techniques based solely on machine learning [66]).

Concerning other important aspects that should be taken into account within a detailed evaluation of programs (see Section 2), our approach, in some cases, can be used for these purposes, too. For example, given that partial equivalence between the student solution and some particular predefined solution has been proven, it confirms that these programs share main characteristics of the used algorithms. Also, our approach can be used for generating failing test-cases (as illustrated in Section 4.1) as a useful feedback for students, as it is done in approaches based on automated testing. Although the main purpose of our approach is to provide high-quality and precise grading at final tests or at competitions, it can be also used as a support for clustering of programs, potentially leading to a finer feedback for specific clusters (e.g. in synergy with approaches for classification and clustering of programs based on static or dynamic analysis [24,55]).

6. Conclusions and further work

There is a significant theoretical and practical progress that has been made recently in the field of regression verification. We have shown that regression verification can be successfully used in automated evaluation of programs at introductory programming courses, more advanced algorithms courses, and programming competitions, and that it can be very useful for both teachers and students (without affecting the teaching methodology itself). Showing equivalence with the teacher solution gives a much higher confidence in the correctness of student program. We find that regression verification should be used as an extension of classical evaluation process. Moreover, for loop-free programs, regression verification may even replace testing, as results obtained for such programs are definite.

The implementation of the proposed approach transforms C/C++ programs and prepares them for regression verification. We have shown that our system LAV can be successfully used in this context. The presented results have shown that our tools were able to automatically show some kind of equivalence for almost all student programs that are equivalent to model solutions (except a few of those that used unmodeled library functions). For loop-free programs, the total equivalence was shown, while for programs with loops, in some cases a very strong relation of partial equivalence was fully automatically shown, and in all other cases k-equivalence was shown. In 10% of the programs from the two considered real world corpora regression verification found bugs that were not previously discovered by testing and automated bug finding. This shows that even when test-cases are carefully manually crafted and achieve complete code coverage of the model solutions, testers fail to predict all possible situations where the student solution might go wrong and bugs can go undetected. Therefore, regression verification can add to quality and precision of automated evaluation, offering an important complement to testing and, in some cases, even a good alternative for a hard and time-consuming job of manually designing test-cases. The precision of the obtained results shows that these techniques could be integrated into a grading system that would be more reliable than those based on testing. We have analyzed functionally non-equivalent solutions and identified that we can get the most from regression verification in situations where solutions have rich branching structure (either imposed by problem definition or artificially introduced by students). We have described some of the most common sources of bugs and we have analyzed and described the types of problems that can be efficiently assessed by regression verification. Uninterpreted functions can be successfully applied to showing partial equivalence in some cases, but not always.

We are planning to introduce other, more powerful, regression verification techniques [20] and also to develop new ones, such that regression verification can be successfully applied to a wider set of problems. We are also planning to improve our tool for automated transformation of programs and to integrate fully automated regression verification into our set of techniques for automated evaluation of students programs.

Acknowledgements. This work was partially supported by the Serbian Ministry of Science grant 174021 and by the COST action CA15123.

References

1. Afzal, W., Torkar, R., Feldt, R.: A systematic review of search-based testing for non-functional system properties. Information and Software Technology 51(6), 957–976 (2009)
2. Ala-Mutka, K.M.: A Survey of Automated Assessment Approaches for Programming Assignments. Computer Science Education 15, 83–102 (2005)
3. Allen, I.E., Seaman, J.: Learning on demand: Online education in the United Statesf. Tech. rep., The Sloan Consortium (2010)
4. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: POPL. pp. 91–102. ACM (2006)
5. Babic, D., Hu, A.J.: Calysto: Scalable and Precise Extended Static Checking. In: ICSE. pp. 211–220. ACM (2008)
6. Backes, J., Person, S., Rungta, N., Tkachuk, O.: Regression verification using impact summaries. In: SPIN'13. pp. 99–116 (2013)
7. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories. In: Handbook of Satisfiability. vol. 185, pp. 825–885. IOS Press (2009)
8. Barthe, G., Grégoire, B., Kunz, C., Lakhnech, Y., Béguelin, S.Z.: Automation in computer-aided cryptography: Proofs, attacks and designs. In: CPP. pp. 7–8 (2012)
9. Beyer, D.: Automatic Verification of C and Java Programs: SV-COMP 2019. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 133–155. Springer (2019)
10. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: OSDI. pp. 209–224. USENIX (2008)
11. Cheang, B., Kurnia, A., Lim, A., Oon, W.C.: On Automated Grading of Programming Assignments in an Academic Institution. Computers and Education 41(2), 121–131 (2003)
12. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: TACAS. pp. 168–176. Springer (2004)
13. Clarke, E.M.: 25 Years of Model Checking — The Birth of Model Checking. Springer (2008)
14. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT (2009)
15. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL. ACM (1977)
16. Dijkstra, E.: Notes on structured programming. EUT report. WSK, Dept. of Mathematics and Computing Science, Technische Hogeschool Eindhoven, 2nd ed. edn. (1970)
17. Douce, C., Livingstone, D., Orwell, J.: Automatic Test-based Assessment of Programming: A Review. Journal on Educational Resources in Computing 5(3) (2005)
18. Ellsworth, C.C., Fenwick, Jr., J.B., Kurtz, B.L.: The Quiver System. In: SIGCSE. ACM (2004)
19. Ertmer, P.A., Richardson, J.C., Belland, B., Camin, D., Connolly, P., Coulthard, G., Lei, K., Mong, C.: Using Peer Feedback to Enhance the Quality of Student Online Postings: An Exploratory Study. Journal of Computer-Mediated Communication 12(2), 412–433 (2007)
20. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: ASE. pp. 349–360. ACM (2014)
21. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. SIGPLAN Not. 37(5), 234–245 (2002)
22. Foundation, F.S.: Using the GNU Compiler Collection: gcov — a Test Coverage Program (1988-2019), `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`
23. GeeksforGeeks: A CS portal for geeks (2019), `https://www.geeksforgeeks.org/`
24. Glassman, E.L., Scott, J., Singh, R., Guo, P.J., Miller, R.C.: Overcode: Visualizing variation in student solutions to programming problems at scale. Comput.-Hum. Interact. 22(2) (2015)
25. Godlin, B.: Regression verification: Theoretical and implementation aspects (2008), masters Thesis, Technion, Israel Institute of Technology
26. Godlin, B., Strichman, O.: Regression verification. In: Proceedings of the 46th Annual Design Automation Conference. pp. 466–471. DAC '09, ACM, New York, NY, USA (2009)

27. Godlin, B., Strichman, O.: Regression verification: proving the equivalence of similar programs. Softw. Test., Verif. Reliab. 23(3), 241–258 (2013)
28. Grivokostopoulou, F., Perikos, I., Hatzilygeroudis, I.: An educational system for learning search algorithms and automatically assessing student performance. International Journal of Artificial Intelligence in Education (1), 207–240 (2017)
29. Gulwani, S., Radiček, I., Zuleger, F.: Feedback generation for performance problems in introductory programming assignments. In: FSE. pp. 41–51. ACM (2014)
30. Guo, P.: Python is the Most Popular Introductory Teaching Language at Top U.S. Uni. (2014)
31. HackerRank: For devs, companies and schools. (2019), `https://www.hackerrank.com`
32. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10) (1969)
33. Huang, L., Holcombe, M.: Empirical investigation towards the effectiveness of Test First programming. Information and Software Technology 51(1), 182–194 (2009)
34. Huang, S.Y., Cheng, K.T.: Formal equivalence checking and design debugging. Kluwer (1998)
35. Ihantola, P., Ahoniemi, T., Karavirta, V., Seppälä, O.: Review of Recent Systems for Automatic Assessment of Programming Assignments. In: Koli Calling. pp. 86–93. ACM (2010)
36. Ihantola, P.: Creating and Visualizing Test Data From Programming Exercises. Informatics in education 6(1), 81–102 (2007)
37. Juniwal, G., Donzé, A., Jensen, J.C., Seshia, S.A.: Cpsgrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory. In: EMSOFT (2014)
38. Kefalas, P., Stamatopoulou, I.: Using screencasts to enhance coding skills: The case of logic programming. Comput. Sci. Inf. Syst. 15(3), 775–798 (2018)
39. King, J.C.: Symbolic Execution and Program Testing. Commun. ACM 19(7) (1976)
40. Kop, C., Nishida, N.: Automatic constrained rewriting induction towards verifying procedural programs. In: Programming Languages and Systems, LNCS, vol. 8858. Springer (2014)
41. Krusche, S., Seitz, A.: Artemis: An automatic assessment management system for interactive learning. In: Proceedings of the 49th ACM Technical Symposium on Computer Science Education. pp. 284–289. SIGCSE '18, ACM (2018)
42. Kulkarni, C., Wei, K.P., Le, H., Chia, D., Papadopoulos, K., Cheng, J., Koller, D., Klemmer, S.R.: Peer and self assessment in massive online classes. Comput.-Hum. Interact. 20(6) (2013)
43. Laski, J., Stanley, W.: Software Verification and Analysis: An Integrated, Hands-On Approach. Springer, 1 edn. (2009)
44. Lattner, C.: The LLVM Compiler Infrastructure (2012), `http://llvm.org/`
45. Li, S., Xiao, X., Bassett, B., Xie, T., Tillmann, N.: Measuring code behavioral similarity for programming and software engineering education. In: ICSE (2016)
46. Luxton-Reilly, A., Simon, Albluwi, I., Becker, B.A., Giannakos, M., Kumar, A.N., Ott, L., Paterson, J., Scott, M.J., Sheard, J., Szabo, C.: Introductory programming: A systematic literature review. In: Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education. pp. 55–106. ITiCSE 2018, ACM (2018)
47. Mandal, A.K., Mandal, C.A., Reade, C.: A System for Automatic Evaluation of C Programs: Features and Interfaces. IJ. of Web-Based Learning and Teaching Technologies 2(4) (2007)
48. Marin, V.J., Pereira, T., Sridharan, S., Rivero, C.R.: Automated personalized feedback in introductory java programming moocs. In: 2017 IEEE 33rd International Conference on Data Engineering (ICDE). pp. 1259–1270 (2017)
49. McCabe, T.: Structured testing. Tutorial Texts Series, IEEE (1983)
50. Miguel A. Revilla: Uva online judge. (1995-2019), `https://uva.onlinejudge.org/`
51. Mike Mirzayanov: Codeforces (2010 - 2019), `http://codeforces.com/`
52. Moghadam, J.B., Choudhury, R.R., Yin, H., Fox, A.: Autostyle: Toward coding style feedback at scale. In: ACM Conference on Learning @ Scale. pp. 261–266. L@S, ACM (2015)
53. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. Wiley Publishing, 3rd edn. (2011)
54. Naudé, K.A., Greyling, J.H., Vogts, D.: Marking Student Programs Using Graph Similarity. Computers and Education 54(2), 545–561 (2010)

55. Nguyen, A., Piech, C., Huang, J., Guibas, L.: Codewebs: Scalable homework search for massive open online programming courses. In: WWW. pp. 491–502. ACM (2014)
56. Pappano, L.: The year of the MOOC (2012)
57. Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., Paterson, J.: A Survey of Literature on the Teaching of Intr. Prog. In: WG reports on ITiCSE. ACM (2007)
58. Pieterse, V.: Automated assessment of programming assignments. In: CSERC (2013)
59. Post, H., Sinz, C.: Proving functional equivalence of two AES implementations using bounded model checking. In: ICST. pp. 31–40 (2009)
60. Rivers, K., Koedinger, K.: Automating hint generation with solution space path construction. In: 12th Intl. Conf. on Intelligent Tutoring Systems (2014)
61. Rizzardini, R.H., Garca-Pealvo, F.J., Kloos, C.D.: Massive open online courses: Combining methodologies and architecture for a success learning. JUCS 21(5), 636–637 (2015)
62. Scheben, C., Schmitt, P.H.: Efficient self-composition for weakest precondition calculi. In: FM 2014: Formal Methods, LNCS, vol. 8442, pp. 579–594. Springer (2014)
63. Sedgewick, R., Wayne, K.: Algorithms. Addison-Wesley Professional, 4th edn. (2011)
64. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: PLDI. pp. 15–26. ACM (2013)
65. Sphere Research Labs: Sphere Online Judge (SPOJ) (2019), `http://www.spoj.com/`
66. Srikant, S., Aggarwal, V.: A system to grade computer programming skills using machine learning. In: ACM KDD. pp. 1887–1896. ACM (2014)
67. Strichman, O., Godlin, B.: Regression verification - a practical way to verify programs. In: VSTTE. LNCS, vol. 4171, pp. 496–501. Springer (2005)
68. Strichman, O.: Special issue: program equivalence. Formal Methods in System Design 52(3), 227–228 (Jun 2018), `https://doi.org/10.1007/s10703-018-0318-y`
69. Stuikys, V., Burbaite, R., Damasevicius, R.: Teaching of computer science topics using metaprogramming-based glos and LEGO robots. Informatics in Education 12(1), 125–142 (2013)
70. Taherkhani, A., Korhonen, A., Malmi, L.: Automatic recognition of students' sorting algorithm implementations in a data structures and algorithms course. In: Koli Calling. ACM (2012)
71. Tillmann, N., Halleux, J.: Pex – White Box Test Generation for .NET . In: TAP. LNCS, vol. 4966, pp. 134–153. Springer (2008)
72. Topcoder: Topcoder (2001–2019), `https://www.topcoder.com/`
73. Valiente, J.A.R., Merino, P.J.M., Díaz, H.J.P., Ruiz, J.S., Kloos, C.D.: Evaluation of a learning analytics application for open edX platform. Comput. Sci. Inf. Syst. 14(1), 51–73 (2017)
74. Verdoolaege, S., Palkovic, M., Bruynooghe, M., Janssens, G., Catthoor, F.: Experience with widening based equiv. checking in realistic multimedia systems. J. Elec. Testing 26(2) (2010)
75. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Eng. 10(2), 203–232 (2003)
76. Vujošević Janičić, M.: LAV (2009 -), `http://argo.matf.bg.ac.rs/?content=lav`
77. Vujošević Janičić, M., Kuncak, V.: Development and Evaluation of LAV: An SMT-Based Error Finding Platform. In: VSTTE. pp. 98–113. LNCS, Springer (2012)
78. Vujošević Janičić, M., Nikolić, M., Tošić, D., Kuncak, V.: Software verification and graph similarity for automated evaluation of students assignments. Inf. and Soft. Tech. 55(6) (2013)
79. Vujošević Janičić, M., Tošić, D.: The Role of Programming Paradigms in the First Programming Courses. The Teaching of Mathematics XI(2), 63–83 (2008)
80. Welsch, Y., Poetzsch-Heffter, A.: A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries. Sci. Comput. Program. 92, 129–161 (2014)

Milena Vujošević Janičić is currently an assistant professor at the Department of Computer Science, Faculty of Mathematics, University of Belgrade. Her main research interests are in automated bug finding, model checking and application of software verification techniques in different fields. She is a member of the Automated Reasoning GrOup (ARGO) at the University of Belgrade.

Filip Marić is currently an associate professor at the Department of Computer Science, Faculty of Mathematics, University of Belgrade. His main research interests are in interactive theorem proving and its applications in formalization of mathematics and software verification. He is also interested in SAT and SMT solving and their applications and in teaching programming at introductory level. He is a member of the Automated Reasoning GrOup (ARGO) at the University of Belgrade.

Received: December 20, 2018; Accepted: July 10, 2019.