Evaluating Test Suite Effectiveness and Assessing Student Code via Constraint Logic Programming

Kyle Dewey
UC Santa Barbara
kyledewey@cs.ucsb.edu

Phill Conrad
UC Santa Barbara
pconrad@cs.ucsb.edu

Michelle Craig
University of Toronto
mcraig@cs.toronto.edu

Elena Morozova
UC Santa Barbara
emorozova@umail.ucsb.edu

ABSTRACT

A good suite of test inputs is an indispensable tool both for manual and automated assessment of student submissions to programming assignments. Yet, without a way to evaluate our test suites, it is difficult to know how well we are doing, much less improve our practice. We present a technique for evaluating a hand-generated test suite by comparing its ability to find defects against that of a test suite generated automatically using Constraint Logic Programming (CLP). We describe our technique and present a case study using student submissions for an assignment from a second-year programming course. Our results show that a CLP-generated test suite was able to identify significant defects that the instructor-generated suite missed, despite having similar code coverage.

ACM Reference format:
Kyle Dewey, Phill Conrad, Michelle Craig, and Elena Morozova. 2017. Evaluating Test Suite Effectiveness and Assessing Student Code via Constraint Logic Programming. In *Proceedings of ITiCSE'17, July 03-05, 2017, Bologna, Italy.*, , 6 pages.
DOI: http://dx.doi.org/10.1145/3059009.3059051

1 INTRODUCTION

To evaluate student submissions to programming assignments, we need a good set of test cases. This is certainly true when assessing student code by hand, and even more crucial when assessment is automated. As course enrollments grow, so has the use of automated assessment tools, sometimes called *autograders* [8, 20]. An ideal test suite for either manual or automated grading would be able to (a) provide helpful information on in-progress student work to detect and diagnose defects, and (b) differentiate between student solutions in a way that maps to student learning so that instructors can assign an appropriate distribution of grades.

In practice, however, instructors have limited time to devote to developing test suites for their course assignments. Moreover, instructors often update assignments over time, be it for improving the assignment, altering learning objectives, preventing previous solutions from being dishonestly proliferated, or otherwise. The end result is that gaps can be easily introduced in a test suite, leading to overlooked edge cases and untested behaviors. By their very nature, these gaps are elusive and difficult to pinpoint, as tests fundamentally can only spot problems one is looking for.

We observe that the problem of developing better test suites and testing techniques is well-studied in Software Engineering, with a variety of automated techniques being employed. These approaches have found several thousand bugs in popular software like `gcc`, `clang`, and Mozilla FireFox [6, 11–13, 21]. Most importantly, these bugs have been found in software which has been heavily tested manually with extensive handwritten test suites. As such, these sort of automated testing techniques are well-suited to our problem of finding gaps in a traditional handcrafted test suite.

Building on this research, we applied an automated testing technique based on *Constraint Logic Programming* (CLP [9], explained in Section 3) to the automated generation of a test suite for a programming assignment (described in Section 4) from a Sophomore-level Java course. The assignment is complex, requiring students to implement non-trivial modifications to the tokenization, parsing, and evaluation components of an interpreter for infix arithmetic expressions. Moreover, the assignment has a history of modification and it features a large handwritten test suite authored jointly by two course instructors. For these reasons, it serves as an excellent case study in seeing where the gaps lie in its existing tests.

This case study (described in Section 5) has revealed a number of deficiencies in the handwritten tests, with a multitude of defects being found by the automated tests which were not spotted by the manual tests. We confirmed via manual code inspection that these defects can be tied to real bugs in student code which were missed by the handwritten test suite. We also found that the tests passed and failed by the automated test suite were instrumental in grouping student solutions with similar behaviors.

The contributions of this paper are (1) a description of how to apply CLP to automatically produce a test suite for an interpreter's tokenizer, parser, and evaluator components (2) a technique for evaluating a handwritten test suite's defect-finding effectiveness by comparison against an automatically-generated test suite (3) a case study of this technique for a non-trivial sophomore-level Java assignment, demonstrating its effectiveness.

This work was supported by NSF CCF-1319060.

ITiCSE'17, July 03-05, 2017, Bologna, Italy.
© 2017 ACM. ISBN 978-1-4503-4704-4/17/07…$15.00.
DOI: http://dx.doi.org/10.1145/3059009.3059051

2 RELATED WORK

At many campuses, Computer Science course enrollments are increasing faster than resources available to instructors; we are being asked to evaluate more student work in the same amount of time. Wilcox et al. [20] argues that "carefully designed and managed automation can improve student performance while realizing a significant savings of scarce teaching resources."

Tillmann et al. [17] created Pex4Fun, which is a non-traditional solution to grading in Massive Open Online Courses. Pex4Fun incorporates automated grading based on symbolic execution. Gulwani et al. [5] attempt to address the tedious process of manually creating tests by developing a tool that generates feedback on the performance of student programs. The tool is a light-weight programming language extension that uses dynamic analysis. Singh et al. [15] devised a new method for giving feedback to incorrect student solutions. They created a language which describes correction rules for a specific student solution. Their method requires that the instructor provide a reference implementation of the assignment solution. Ihantola [7] applied an automatic test data generation tool named *Java Pathfinder* [18] to automatic assessment and found that symbolic execution with lazy initialization can be used to generate test data directly from student programs.

CLP was first introduced in Jaffar et al. [9], as a general framework encompassing logic programming languages such as Prolog [16]. For the purposes of this paper, CLP is viewable fundamentally as Prolog with some convenient extensions. The work of Dewey et al. [1–3] discusses how CLP can be applied to generating test suites for a variety of domains, including language interpreters [2], data structure APIs [1], and typecheckers [3]. While Dewey et al. uses CLP for the purposes of testing, a significant amount of CLP code must be written for the particular domain which is to be tested. While CLP has shown itself to be effective for testing large industrial applications, we are not aware of any work describing its application to evaluating student code, nor has it specifically been applied to testing tokenizers or parsers.

3 TEST SUITE GENERATION VIA CLP

This section discusses why and how we use CLP for generating test suites for language tokenizers, parsers, and evaluators. We start first with some background on CLP-based test suite generation.

3.1 Background on CLP-Based Testing

Programming languages allow one to use code to specify how outputs should be derived from inputs, and in this respect, CLP is no different from more traditional languages.

What makes CLP interesting for our purposes is that, intuitively, this same derivation process can be run in *reverse*, as well as being run with no initial inputs or outputs whatsoever. That is, CLP can be used *both* to derive the inputs corresponding to some given outputs, as well as derive entire valid input/output pairs. From a high level, this is possible with CLP because CLP code describes a series of logical constraints relating inputs to outputs. These constraints can be efficiently explored via a multitude of existing CLP engines (e.g., SWI-PL [19] and GNU Prolog [4]), allowing for the generation of whole input/output pairs which satisfy the constraints. The technical details behind how this is possible is beyond the scope of this discussion, but it has been well-described elsewhere (e.g., [9, 16]).

$$\begin{aligned} \mathit{Exp} &::= \mathit{AddExp} \\ \mathit{AddExp} &::= \mathit{PrimExp} \;\mid\; \mathit{PrimExp}\ \text{'-'}\ \mathit{AddExp} \\ \mathit{PrimExp} &::= \text{'0'} \;\mid\; \text{'1'} \;\mid\; \text{'('}\ \mathit{Exp}\ \text{')'} \;\mid\; \text{'-'}\ \mathit{PrimExp} \end{aligned}$$

Figure 1: Small grammar used for running CLP example.

This capability of CLP to derive valid input/output pairs of a program lies at the heart of CLP-based testing (e.g., [1–3]). For our specific purposes, from a high level this entails first implementing a *reference solution* in CLP, which only differs from a traditional reference solution in the fact that CLP is used as the implementation language. The CLP-based reference solution can then be used to derive valid input/output pairs which will stress different parts of the solution. In a straightforward manner, these input/output pairs can then be applied to testing student solutions, where the input in a pair specifies a test input, and the output of a pair specifies the expected test result. A specific example of how this is done for a tokenizer is presented in Section 3.2.

The aforementioned strategy has one weakness, however: we will never produce a test with no output, as when an invalid input has been passed to the program. It is not possible to derive invalid inputs via this mechanism, as by definition there is no way to relate them to valid outputs. For our purposes, this is unfortunate, as invalid inputs are useful for testing that student error-checking routines exist and behave as intended; therefore a different approach is needed for generating invalid inputs.

To this end, we employ *mutation-based fuzz testing* (e.g, [6]), which starts with arbitrary valid inputs. These valid inputs are then *mutated* in some way by intentionally injecting something which is guaranteed to make it invalid. The invalid inputs produced are suitable for testing, and all should trigger error-checking routines in the student solutions. While more sophisticated approaches are possible (e.g., generating invalid inputs by construction [3]), the approach used here is both simple and effective for finding faults in student solutions. Further discussion of how this is done for a tokenizer and a parser is presented in Section 3.3.

3.2 Generating Valid Inputs

This subsection discusses how we use CLP to test language tokenizers, parsers, and evaluators. Throughout this subsection, we use a running example based on the grammar shown in Figure 1, with the corresponding tokens `0` (zero), `1` (one), `-` (minus), `(` (left parentheses) and `)` (right parentheses). While this grammar is admittedly simple, it serves to illustrate all the applicable core concepts, and it forms a subset of the grammar used in the case study.

An executable CLP-based tokenizer applicable to tokenizing the grammar in Figure 1 is presented in Figure 2. The `charToToken` helper procedure simply maps characters to their token representations. The `tokenize` procedure in Figure 2 consists of two *rules*. The first rule states that if there are no characters to tokenize, then there are no tokens produced. The second rule states that if the

```
% charToToken: Character, Token
charToToken('0', token_zero).
charToToken('1', token_one).
charToToken('-', token_minus).
charToToken('(', token_lparen).
charToToken(')', token_rparen).

% tokenize: Characters, Tokens
tokenize([], []).            % tokenize rule 1
tokenize([SingleChar|Chars], % tokenize rule 2
         [SingleToken|Tokens]) :-
    charToToken(SingleChar, SingleToken),
    tokenize(Chars, Tokens).
```

Notation: % indicates end of line comments. [] is empty list. [A|B] is a list, with head A and rest of list B. tokenize takes two parameters: a list of characters, and a list of tokens produced.

Figure 2: CLP-based tokenizer for language of Figure 1.

character input begins with a single character, then the tokens produced begin with a single token, where the token is derived from the charToToken helper procedure. Furthermore, the second rule recursively calls tokenize to process the rest of the input. While we have kept this illustrative example as simple as possible, it is straightforward to add CLP code to allow for whitespace, integers with multiple digits, and multiple-character tokens, all of which appear in the actual assignment used in our case study.

When given a valid input list of characters, the tokenize routine behaves as follows (where lines starting with ?- indicate something typed by the user at a prompt for a typical CLP engine):

```
?- tokenize(['1', '-', '0'], Tokens).
Tokens = [token_one, token_minus, token_zero].
?- tokenize(['(', '-', '1', ')'], Tokens).
Tokens = [token_lparen, token_minus, token_one,
          token_rparen].
```

With an invalid input, the engine instead returns false, indicating the input could not be successfully tokenized. For example, the engine cannot tokenize the input below, because + is not a valid token according to the language in Figure 1:

```
?- tokenize(['1', '+', '1'], Tokens).
false.
```

As previously discussed, this CLP-based solution can be used to generate input/output pairs. For example, the code below will derive all inputs of length 4, along with their corresponding lists of tokens:

```
?- length(Input, 4), tokenize(Input, Tokens).
Input = ['0', '0', '0', '0'],
Tokens = [token_zero, token_zero, token_zero,
          token_zero] ... % many more elided
```

This capability to automatically generate valid inputs lies at the heart of CLP's power for test case generation.

The parser for these tokens using a standard recursive-descent style can similarly be implemented in CLP, as well as a typical pre-order-based recursive expression evaluator. Because these components do not significantly differ in style from the aforementioned tokenizer example, they have been omitted for space reasons.

3.3 Generating Invalid Inputs

For generating invalid inputs for the tokenizer, we insert characters which will never yield valid tokens into an otherwise tokenizable stream of characters. Specifically, we insert $, = (ensuring it does not follow either > or <), =>, and =<. The character $ was chosen arbitrarily as a representative of an unconditionally invalid character, and the rest of the characters were chosen as they intuitively seem more likely to trigger faults in a buggy tokenizer. For generating invalid inputs for the parser, we first produce a valid list of tokens which can be parsed to form a valid expression. We then insert an arbitrary valid token into the list, either an integer 0 or 2 (arbitrarily chosen), or any other one of a finite list of remaining valid tokens. Because this process may still yield a parsable list of tokens (as when negating a subexpression), we run the CLP-based parser on the newly generated input to ensure that it **fails**, thus ensuring the input is invalid. While these approaches to generating invalid inputs for the tokenizer and parser are simplistic, we have nonetheless found them to be effective at finding faults in student code.

As for the evaluator, relatively few things act as invalid inputs. By construction, the AST definition in both the Java and CLP reference solutions does not allow for the construction of ASTs which are in any way invalid. With this in mind, the only significant edge case which can be safely deemed “invalid” is that of cases which trigger division by zero, which is supposed to be checked beforehand by student solutions. We observed that a significant number of the generated valid parser outputs (ASTs produced as described in Section 3.2) would attempt to perform division by zero. As such, if we simply re-used these ASTs as inputs to the evaluator, along with a record of what the AST should evaluate to (be it a number or a trigger for division by zero).

4 THE PROGRAMMING ASSIGNMENT

The programming assignment used as a case study was given in a second-year programming course in advanced application programming. Students were given code for a working interpreter of infix arithmetic expressions, with separate Java classes for (1) a finite-state-automaton based tokenizer (and classes for various kinds of tokens), (2) a recursive descent parser that corresponded exactly to the grammar given and produced an Abstract Syntax Tree (AST) (and classes for various AST nodes) (3) a straightforward interpreter based on a pre-order traversal of the AST.

To simplify the assignment, it was assumed that all constants and expressions would be of type integer. The given code was capable of interpreting expressions involving addition (+), subtraction (-), multiplication (*), integer division (/), unary minus (-), and parentheses (()). The students were also given a grammar in EBNF for the language supported by the interpreter, shown in Figure 3. The students were then required to add support for six relational operators, (<, <=, >, >=, ==, !=) (each of which returns either 0 or 1 based on the truth value of the comparison), as well as exponentiation (**), as reflected in the modified grammar of Figure 4.

```
expr ::= add-expr
add-expr ::= mult-expr ( ( '+' | '-' ) mult-expr ) *
mult-expr ::= primary ( ( '*' | '/' ) primary ) *
primary ::= '(' expr ')' | INTEGER | '-' primary
```

Figure 3: Given EBNF grammar

```
expr ::= comp-expr
comp-op ::= '==' | '!=' | '<' | '<=' | '>' | '>='
comp-expr ::= add-expr ( comp-op add-expr ) *
add-expr ::= mult-expr ( ( '+' | '-' ) mult-expr ) *
mult-expr ::= exp-expr ( ( '*' | '/' ) exp-expr ) *
exp-expr ::= primary '**' exp-expr | primary
primary ::= '(' expr ')' | INTEGER | '-' primary
```

Figure 4: Modified EBNF grammar

The intent of requiring students to add this particular set of new features was that they necessitated the students to be able to handle two cases that are not in the starting code: (1) tokens involving multiple characters—especially tokens where one valid token is a prefix of another valid token (e.g. `<` prefix of `<=`, `*` prefix of `**`), (2) a right-associative binary operator (exponentiation); all binary operators in the starting code are left-associative.

5 THE CASE STUDY

During Fall 2016, students submitted solutions which were then autograded via a traditional handcrafted test suite composed of 230 tests. The assignment had an option for either individual or pair submission. The study is based on 48 submissions where either the sole author or both partners gave informed consent.

We then used the CLP-based technique described in Section 3 to generate a test suite composed of 7,291,812 tests, specifically tests focused on the tokenizer, parser, and evaluator for the grammar from Figure 4. Where possible, the same test input was reused to test multiple components. For example, consider the following test input:

```
1 - 1
```

This input should tokenize, parse, and evaluate successfully down to the value `0`. As such, it can be used as a test each component individually; that is, the characters serve as a tokenizer test, the tokens corresponding to it serve as a parser test, and the expression produced by the parser serves as an evaluator test. If a solution failed part of the tokenizer to parser to evaluator chain, a correct intermediate form was substituted so the remaining components could be individually tested.

We first ran both test suites on the instructor's own Java reference solution. The CLP code corresponding to the Java reference solution was intentionally **not** coded by the same individual, to reduce the likelihood of the Java and CLP reference solutions sharing common bugs. The instructor's solution passed 100% of both the hand-crafted tests and the CLP-generated tests, indicating that the CLP solution was correct and not marking any outputs incorrectly as valid or invalid.

We then ran both the hand-crafted and the CLP-generated tests against the student solutions. Our raw data, therefore, consisted of results for 230 hand-crafted tests and just over seven million CLP-generated tests for each of the 48 student solutions. Looking at the data, the following conclusions were immediately reached:

- There was no case where a solution passed all of the CLP-based tests, but failed at least one of the handcrafted tests.
- Of the 40 solutions that passed all of the 230 handcrafted tests, only 30 of those passed all 7,291,812 of the CLP-based tests. This alone was a clear indication that the CLP-based suite detected at least one defect that the handcrafted tests did not.

Eight of the solutions failed tests on both the handcrafted test suite and the CLP-based test suite. Because failures occurred on both test suites for these solutions, a more sophisticated approach was necessary in order to draw any meaningful conclusions.

5.1 Test Suite Comparison via Equivalence Classes

We observe that solutions can be separated into equivalence classes based on the tests they fail. That is, it is common for multiple solutions to fail the exact same set of tests, suggesting that different solutions share the same underlying defects.

We first partitioned the 48 solutions on the basis of which tests were failed on the handcrafted tests, which yielded only six equivalence classes. These six classes are shown visually in the top half of Figure 5. There were:

- three singleton classes
- one class of two solutions
- one class of three solutions
- one class of 40 solutions—these are the solution that failed none of the hand-crafted tests

While the small number of equivalence classes may raise the question of plagiarism, follow-up with MOSS [14] showed that plagiarism alone cannot account for this effect.

Partitioning the 48 solutions based on which tests were failed by the CLP-based test suite resulted in a a further refinement of these six equivalence classes were further refined into thirteen classes, as shown in the bottom half of Figure 5. The partitions produced via the CLP-based test suite are represented visually by the arrows in Figure 5. Each division represents a case where the CLP-based tests are potentially revealing something that the manual test suite missed.

The set of solutions that passes all of the CLP tests contains only 30 solutions; ten of the solutions from the original equivalence class of 40 failed some of the CLP-based tests, resulting in four additional equivalence classes. The original classes of two and three were also further split into singletons. Each of these splits represents a case where the CLP-based tests were able to more distinguish among solutions with finer granularity, thus potentially revealing additional defects missed by the manual test suite.

5.2 Code Inspection

We did a manual code inspection of representative solutions from both sides of each equivalence class split to learn more about what these splits signified. It is tempting to assume that each such split indicates a new bug or set of bugs. Our explorations show that

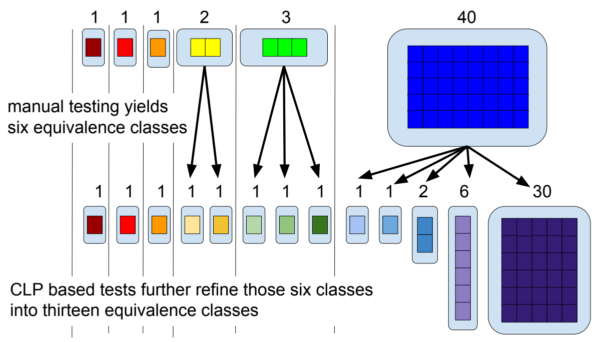
1
1
1
2
3
40
manual testing yields
six equivalence classes
1
1
1
1
1
1
1
1
1
1
2
6
30
CLP based tests further refine those six classes
into thirteen equivalence classes

Figure 5: Equivalence classes of student submissions, based on which tests they failed.

this is often, though not always the case. The other possibility is that the solutions have made similar errors but errors that differ enough in the particular way they are incorrect, such that they pass different numbers of tests. However, each division did offer some insight into ways in which students approached the problem, and the types of mistakes they made in their code.

The original large equivalence class of 40 students is broken up five ways, overall revealing that 10 students who passed all the handcrafted tests nonetheless still had bugs in their code (an indication that if the test suite had been more powerful, there may have been more rigor and fairness in the grades assigned). These 10 were divided into two singletons, a pair, and a group of 6. One of the singletons was a solution that, although it was correct from the standpoint of an end user, failed many parser tests because the testing code relied on the `.equals()` method of one of the student-defined AST classes to work properly (when comparing actual vs. expected results), however the student failed to override this method. This was a case where the handcrafted test suite was deficient. The three remaining classes were all characterized by various errors involving negative exponents, bringing to light the fact that the instructors had completely overlooked testing for the cases of zero and negative exponents (focusing the manual tests instead on the right-associativity of the operator).

Students that correctly handled the exponent operator used a variety of approaches. Some students computed a value using Java's `Math.pow()` method and then cast the result to an integer value. Others used loops that did repeated multiplication for non-negative exponents, and repeated division for negative exponents. With these loops, some handled x^0 as a special case while others initialized a product value to 1, and then used a loop to repeatedly multiply by the base of the exponent (so that zero iterations of the loop naturally returns the correct value).

Figures 6, 7, and 8 show three incorrect approaches to computing exponents with loops all taken from the group of 40, each passing a different number of CLP-based tests. Figure 6 correctly computes positive and zero exponents, and has an incorrect attempt at negative exponents. Figure 7 correctly calculates positive and zero exponents, but has no code for negative exponents. Figure 8 calculates only positive exponents correctly.

There were two other equivalence classes that were further partitioned by the CLP-based testing: an equivalence class of two,

```
// calculate left ** right
int result = left;
if (right == 0) {
  return 1;
}
else if (right < 0) {
  for (int i = 0; i < (right * -1)-1; i++) {
    result = result / left;
  }
  return result;
} else {
  for (int i = 0; i < right-1; i++) {
    result = result * left;
  }
  return result;
}
```

Figure 6: Incorrect approach to negative exponents

```
// calculate base**exp
int result = 1;
for (int i=0; i<exp; i++)
  result *= base;
return result;
```

Figure 7: Does not handle negative exp

```
// calculate left ** right
int x = left;
for(int i = 1; i < right; i++) {
  x = x * left;
}
return x;
```

Figure 8: Handles neither 0 nor negative exponents

and one of three, each of which was refined into singletons by the CLP-based testing. Code inspection of the class of two solutions revealed that both of the solutions failed the hand-written tests for both the new comparison operators and the exponentiation operator. What distinguished one solution was an error in the finite state automaton; it failed to recognize that the `*` token was a prefix of the `**` token, and did not set up a state transition from the state for the multiplication operator to the state for the exponentiation operator.

The equivalence class of three solutions was refined further by CLP-generated tests into three singletons. All three solutions shared a common bug related to an improperly structured `if/else`. The first of these had a bug not found in the other two, related to an issue of == vs. `.equals()` for objects. All three had problems related to calculation of exponents, but the third solution handled an exponent of `0` correctly, in contrast to the first two solutions.

Our overall conclusion is that while each division between equivalence classes is an interesting place to look for defects, it is not necessarily the case that each corresponds to a particular discrete bug. This sort of difficulty in defining exactly what "bug" means is a known problem in Software Engineering research (e.g., [10]), and this work makes this apparent in the context of educational assessment.

5.3 Test Suite Code Coverage

While the CLP-based test suite exposed clear deficiencies in the handcrafted test suite, we wondered as to whether or not these deficiencies could have been discovered ahead of time with a more traditional approach. To this end, we measured average code coverage across all students for the two test suites. For the handcrafted test suite, we observed an average line coverage of 77%, an average method coverage of 74%, and an average branch coverage of 62%. For the CLP-based test suite, we observed an improvement of three percentage points for average line coverage (80%), an improvement of nine percentage points for average method coverage (83%), and no improvement in average branch coverage (62%). For all coverage metrics, the relatively low values can be uniformly explained by the presence of debugging-oriented code, as well as methods which are good practice to implement but not directly under test (e.g., `hashCode()`, `toString()`, and `equals()`). It is difficult to fairly prune out such code, because some students did nonetheless use it during testing.

As shown, while the CLP-based test suite tended to get better code coverage than the handcrafted test suite, the improvements are often marginal at best. This leads us to conclude that code coverage can be a misleading measure of a test suite's effectiveness. This motivates the direct measurement of the defect-finding power of a test suite, as can be done through our CLP-based approach.

6 CONCLUSIONS AND FUTURE WORK

Our case study shows that a CLP-based approach that generates millions of test cases can be used to expose weaknesses in a handcrafted test suite for a programming assignment. Sifting through the millions of test cases results might seem daunting, but we have shown that by focusing on the differences among representative solutions from different equivalence classes, considerable insight can be gained into test-suite deficiencies, and student errors.

There are two major limitations to this work which we seek to address in future work: unfamiliarity with CLP programming and scalability. To address the first of these, we propose to develop a targeted Prolog/CLP tutorial for instructors who want to apply the technique, or even a domain-specific language which compiles to CLP, allowing instructors to bypass using CLP directly. As for scalability, the fact that our technique entails generating millions of test cases raises research, engineering, and deployment concerns. In our study, we bypassed these concerns by performing an offline analysis after the course was over. We are working towards an open source software framework for managing the computational requirements of applying this technique at scale, enabling its use for online formative and summative assessment.

REFERENCES

[1] Kyle Dewey, Lawton Nichols, and Ben Hardekopf. 2015. Automated Data Structure Generation: Refuting Common Wisdom. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15).* IEEE Press, Piscataway, NJ, USA, 32–43. http://dl.acm.org/citation.cfm?id=2818754.2818761

[2] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language Fuzzing Using Constraint Logic Programming. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14).* ACM, New York, NY, USA, 725–730. DOI: http://dx.doi.org/10.1145/2642937.2642963

[3] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15).* IEEE Computer Society, Washington, DC, USA, 482–493. DOI: http://dx.doi.org/10.1109/ASE.2015.65

[4] Daniel Diaz and Philippe Codognet. 2000. The GNU Prolog System and Its Implementation. In *Proceedings of the 2000 ACM Symposium on Applied Computing - Volume 2 (SAC '00).* ACM, New York, NY, USA, 728–732. DOI: http://dx.doi.org/10.1145/338407.338553

[5] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2014. Feedback Generation for Performance Problems in Introductory Programming Assignments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014).* ACM, New York, NY, USA, 41–51. DOI: http://dx.doi.org/10.1145/2635868.2635912

[6] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Proceedings of the 21st USENIX conference on Security symposium (Security'12).* USENIX Association, Berkeley, CA, USA, 38–38.

[7] Petri Ihantola. 2006. Test Data Generation for Programming Exercises with Symbolic Execution in Java PathFinder. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006 (Baltic Sea '06).* ACM, New York, NY, USA, 87–94. DOI: http://dx.doi.org/10.1145/1315803.1315819

[8] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 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).* ACM, New York, NY, USA, 86–93. DOI: http://dx.doi.org/10.1145/1930464.1930480

[9] J. Jaffar and J.-L. Lassez. 1987. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '87).* ACM, New York, NY, USA, 111–119. DOI: http://dx.doi.org/10.1145/41625.41635

[10] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014).* ACM, New York, NY, USA, 437–440. DOI: http://dx.doi.org/10.1145/2610384.2628055

[11] [meta] Bugs Found by jsfunfuzz 2006. [meta] Bugs Found by jsfunfuzz. (2006). https://bugzilla.mozilla.org/show_bug.cgi?id=349611

[12] [meta] LangFuzz (Grammar-based Mutation Fuzzer) - JS shell bugs 2011. [meta] LangFuzz (Grammar-based Mutation Fuzzer) - JS shell bugs. (2011). https://bugzilla.mozilla.org/show_bug.cgi?id=676763

[13] Jesse Ruderman. 2007. Introducing jsfunfuzz. (2007). http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/

[14] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. 2003. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03).* ACM, New York, NY, USA, 76–85. DOI: http://dx.doi.org/10.1145/872757.872770

[15] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. *SIGPLAN Not.* 48, 6 (June 2013), 15–26. DOI: http://dx.doi.org/10.1145/2499370.2462195

[16] Leon Sterling and Ehud Shapiro. 1994. *The Art of Prolog (2Nd Ed.): Advanced Programming Techniques.* MIT Press, Cambridge, MA, USA.

[17] Nikolai Tillmann, Jonathan De Halleux, Tao Xie, Sumit Gulwani, and Judith Bishop. 2013. Teaching and Learning Programming and Software Engineering via Interactive Gaming. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13).* IEEE Press, Piscataway, NJ, USA, 1117–1126. http://dl.acm.org/citation.cfm?id=2486788.2486941

[18] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. 2003. Model Checking Programs. *Automated Software Engg.* 10, 2 (April 2003), 203–232. DOI: http://dx.doi.org/10.1023/A:1022920129859

[19] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96.

[20] Chris Wilcox. 2015. The Role of Automation in Undergraduate Computer Science Education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15).* ACM, New York, NY, USA, 90–95. DOI: http://dx.doi.org/10.1145/2676723.2677226

[21] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11).* ACM, New York, NY, USA, 283–294. DOI: http://dx.doi.org/10.1145/1993498.1993532