Using Fast Model-Based Fault Localisation to Aid Students in Self-Guided Program Repair and to Improve Assessment

Geoff Birch
University of Southampton
gb2g10@ecs.soton.ac.uk

Bernd Fischer
Stellenbosch University
bfischer@cs.sun.ac.za

Michael Poppleton
University of Southampton
mrp@ecs.soton.ac.uk

ABSTRACT

Computer science instructors need to manage the rapid improvement of novice programmers through teaching, self-guided learning, and assessment. Appropriate feedback, both generic and personalised, is essential to facilitate student progress. Automated feedback tools can also accelerate the marking process and allow instructors to dedicate more time to other forms of tuition and students to progress more rapidly. Massive Open Online Courses rely on automated tools for both self-guided learning and assessment.

Fault localisation takes a significant part of debugging time. Popular spectrum-based methods do not narrow the potential fault locations sufficiently to assist novices. We therefore use a fast and precise model-based fault localisation method and show how it can be used to improve self-guided learning and accelerate assessment. We apply this to a large selection of actual student coursework submissions, providing more precise localisation within a sub-second response time. We show this using small test suites, already provided in the coursework management system, and on expanded test suites, demonstrating scaling. We also show compliance with test suites does not predictably score a class of "almost correct" submissions, which our tool highlights.

Keywords

Debugging; Model-Based Fault Localisation; Student Code Submissions; Assessment; Self-Training

1. INTRODUCTION

Introductory programming courses typically require instructors to assess a large number of small programs by novice programmers. Many of these programs contain errors, ranging from small mistakes to complete design and implementation failures, reflecting the students' misunderstanding of the task or their solution attempt. With limited resources the best learning outcomes are achieved if students are (in the former case) automatically directed towards the locations of their mistakes to allow self-guided repairs, so that the instructors can focus (in the latter case) on addressing fundamental misunderstandings. However, existing basic assessment tools (such as Ceilidh [2], ASSYST [14], and BOSS [16]) based on simple compilation tests, test suites, or model solutions do not provide the detailed feedback necessary for self-guided repairs and do not support instructors in quickly separating solutions with small mistakes from complete failures. Compilation tests do not give any feedback for syntactically correct programs, test suites can give misleading results if programs contain simple errors that affect a large number of test cases, and model solutions cannot account for the variability of student programs.

In this paper we address these issues and describe a fast and accurate fully automated fault localisation tool for C programs and demonstrate its application to a corpus of student programs. Our tool assumes that the programs are specified only by test suites and returns a list of source code locations where a single assignment fault can be repaired to bring the program into compliance with the entire test suite. Such weak specifications make accurate localisation challenging: spectrum-based fault localisation methods [37] are unable to sufficiently narrow down the list of possible fault locations [24], while more precise model-based methods [10] suffer from high run-times. We have recently presented [3] an intelligent search algorithm to scale up model-based fault localisation, but demonstrated it only on a single localisation benchmark. It is thus an open question whether model-based fault localisation can be exploited in aiding novice programmers, in particular with debugging, and in improving the assessment of their solutions; more precisely, we investigate the following research questions:

RQ1 Does model-based fault localisation scale to large submission databases of real-world student code? Is it fast enough, and does it accurately pin-point fault locations, given typical test suites?

RQ2 Does model-based fault localisation improve over the more common spectrum-based methods?

RQ3 Can model-based fault localisation improve automated grading based on functional correctness over test suites?

Our experiments with real-world student code from an existing submissions database are encouraging. Our tool can indeed accurately pin-point fault locations and so allow students to surmount the final hurdle to completing writing of source code that fully complies with a test suite without requiring advanced debugging skills. The short list of locations where an assignment repair is possible, provided by our tool, can aid students in pinpointing improvements before they resubmit their code. The run-times of our tool are competitive with the common, fast spectrum-based localisation techniques while providing significantly more accurate fault localisation, which is of high value to novice programmers. This also allows our tool to work with existing coursework submission

ITiCSE '16, July 09 - 13, 2016, Arequipa, Peru

ISBN 978-1-4503-4231-5/16/07...$15.00

DOI: http://dx.doi.org/10.1145/2899415.2899433

systems and workflows, for both self-training and grading, even at the scale required by Massive Open Online Courses (MOOCs).

Our tool also provides instructors with automated assistance detecting submissions which would provide perfect compliance with a single assignment edit (e.g. incrementing a counter before returning it as the final value) but fail the test suite. This will allow test suite-based automated graders to identify submissions whose quality is being underestimated by this functional assessment method; repair information also provided by our tool can help instructors manually grading the submission. We show that real-world student submissions which are a single assignment edit away from full compliance with a test suite are not provided with a fair mark by grading against that test suite.

2. RELATED WORK

Novice programmers can be divided into three classes [23, 35]: *stoppers*, who give up; *tinkerers*, who appear to modify their code at random; and *movers*, who are already able to engage with feedback to make progress. Even though the movers already demonstrate debugging skills beyond the average of their cohort, all three classes still need high-quality, novice-friendly debugging feedback to allow a self-guided refinement of their solutions towards a correct answer. Enhanced (syntax) error messages appear to be ineffectual in this respect [8], but when students are given hints regarding failures over a test suite, their effort increases compared to students not given hints [6]. On-demand programming feedback provides students the motivation to iterate on submissions [20]. Moreover, a correlation has been found between sessions where students were provided with feedback from a test suite and sessions that improved the student's score [31], indicating this assisted self-training. However, despite its benefits, students are somewhat resistant to a pure test-driven development (TDD) approach [5], as for example embodied in the Web-CAT [9] submission system, partly because TDD also requires expertise in developing test suites as a prerequisite to demonstrating programming ability.

A significant part of the debugging process is finding the location where the file needs to be changed to repair the fault [34]. Tools like AskIgor [38] provide cause chains to assist programmers in localising faults from test failures. While this lowers the threshold, it still requires too much debugging expertise for novices. Our tool instead provides a list of locations where a single assignment fault can be repaired to bring the program into compliance with the entire test suite. Giving tinkerers viable repair locations massively reduces the mutation space they are exploring. Giving movers viable repair locations directs their effort, allowing faster debugging times [22]. This should increase the chance that both will realise the repair and convert a program failure into a learning event.

For programmers to retain their status of movers, they must be provided with feedback in terms of suitable scaffolding to overcome progress stalls and achieve an otherwise unattainable goal [36]. Tinkerers who are unable to be scaffolded into movers will eventually become stoppers due to lack of progress [35]. Since the absolute number of locations shown before the actual repair site is critical to allowing progress before programmer interest drop-off generates stoppers [22], the provided list of locations to investigate must be short. Unfortunately, widespread spectrum-based fault localisation methods are unable to provide sufficiently short candidate lists. Pham et al. [24] tested Tarantula and Ochiai tools and on three variants of an algorithm, with an average length of 13 statements, they found over 9 statements shared the same top suspiciousness ranking. Localisation thus becomes a function of luck.

However, the feedback cannot be too prescriptive or too detailed. Complete program synthesis repair tools such as Auto Grader [29] remove all debugging or repair self-training from the process, trivialising the contribution and so learning of the student. Our tool bridges the gap between standard error reports, which many students lack the expertise to use, and fully automated repair suites.

Automated submission and assessment systems deliver immediate feedback to students for self-training, which is enjoyed by students and feeds into improvements throughout courses that use it [17]. They include tools to help accelerate grading of final submissions, providing both self-training and grading benefits [32]. Such assessment tools take many forms. Ceilidh [2] uses regular expressions to specify the test output of compiled student code, providing more freedom to define the expected answers than traditional test suites. ASSYST [14] contains style and complexity analysis tools that provide metrics beyond test suite correctness, and evaluate non-functional aspects such as code quality and efficiency. FrenchPress [4] focuses on analysis of code style and design, attempting to isolate errors only made by novices in Java programming. AutoGrader [12] requires students to program to a public interface, allowing whitebox testing at the cost of restricting the form of the code submissions. Pex4Fun [33] uses automated test case generation to guide student submissions and to scale to MOOC capacities. This approach requires students to demonstrate advanced debugging skills. Scheme-Robo [28] compares the program source to a model answer, assuming that Scheme's functional design limits the variability of meaningfully different programs that correctly answer the question.

Student code exhibits a wide range of program forms and performance characteristics [30, 21, 18, 26] and students do not necessarily work towards solutions that are similar to an instructor-authored model answer. It is thus inadvisable to only provide feedback directing students towards model answers. Vujošević-Janičić et al. [15] use a static verifier as well as test suites when student solutions do not match the control flow of the model answer. A weighting system then mixes these feedback components to guide students to repair code errors and transform the structure towards that of the model answer or to grade the submission. ASys [13] attempts to overcome the limitation of functional testing using test suites with a customisable semi-automated grading system. Marking templates describe extensions to the test suite for a question, generating new tests specific to the student submission that verify each assessment property defined in the template. In their use on campus, 48% of the grading work was automatically handled by the tools. Interestingly, most of the marks difference between the group using ASys and the control group going through traditional grading was accountable to errors made in the manual marking process. Similarly, our tool provides added feedback beyond functional grading using test suites, identifying student programs which "almost" conform to the question requirements and can be fixed with a single assignment edit. Our tool can also enrich existing systems like ASys, providing this alternative test of closeness to compliance when using test suites in other contexts.

Instructors spending several minutes per student per submission for feedback [19] cannot scale to MOOC environments with thousands of students per course. To test the scaling of an automated hint system to these environments, AutoTeach [1] provides students with access to a hint system which uncovers partial model answers to assist with learning. Such systems, while scaling very well, strictly constrain students to working towards a single model answer. However, the deployment of test suite grading for MOOCs is gaining traction [25] with the drawbacks of formatting issues (3.2% of submissions were awarded zero marks due to formatting issues, not functional failures) and failure to detect almost-correct submissions weighed against the benefits of providing some form

```
/* ... */
int v = a*b;
/* ... */
assert(o==d); //added spec
exit(0);
```

```
int t = symbolic();
/* ... */
int v = (t==5) ? symbolic() : a*b;
/* ... */
assert(o==d);
exit(0);
```

```
int t = symbolic();
/* ... */
int v = (t==5) ? symbolic() : a*b;
/* ... */
assume(o==d);
assert(0); exit(0);
```

Figure 1: Program Transform Process: (a) Code with Test Case Spec; (b) Model of Code; (c) Inverted Model

of code assessment, as manual grading is not viable. Such issues can be reduced via importing more advanced tools into the grading system, assuming their run-time cost can be kept low enough for MOOC execution.

3. DATA SET

We use a data set of around 30k passing and 150k failing Java and Python programs collected by the automated assessment system of the University of Auckland [26]. These programs were written by students to answer 1693 different Computer Science coursework questions. We translated the programs into a C-representation using a simple converter that was designed to retain the style and functionality of programs without translating any detailed differences between the languages, such as language-specific handling of integer overflows. This translation would allow the mapping back of locations to the original Java or Python source code, although our model-based downstream components currently only reason within the specifications of the C programming language they have been translated into. We consider the translated programs in C to be the corpus on which we are testing the efficacy of our tool. Our methods are more directly applicable for native reasoning on Java or Python programs by using suitable downstream components designed for those languages.

We rejected programs that produced translation failures (typically due to unsupported standard library calls), that use floating-point arithmetic (which is not yet supported for symbolic exploration by our downstream components), that comprise less than three assignments (to avoid trivial localisation tasks), or that Klee or GCov (which we use as downstream components) could not process. We then analysed the data set to identify pairs where a student had submitted a failing program which, after a single assignment edit, was later found in the database passing the full test suite. This yielded 304 pairs. These programs contain an average of 5 assignments in 11 statements (as counted by GCov).

We ran a script to analyse each provided test suite (averaging 7.8 tests per pair) and generated new test suites that randomly picked inputs within an order of magnitude of the existing values. The passing program from the code pair was used as oracle to establish desired output values. These large test suites average 154 tests per pair, with 80 failing.

4. METHOD

Model-based fault localisation [27] is the application of model-based diagnosis methods [7] to programs. It involves three main steps: (i) the construction of a logical model from the original program; (ii) the symbolic analysis of this model; and (iii) mapping any faults found in the model back to program locations. The approach to model-based fault localisation used by our tool is to transform the program so that a symbolic program verification tool can be reused for all three steps.

Our tool is fed a C program and a set of failing test cases written in the form of an input and a desired output. The worked example of Figure 1 gives excerpts from a C program that, for the fifth assignment, declares an int `v` equal to `a*b` and eventually assigns the final value to int `o`.

Each test case is taken as a search branch to be explored for potential assignment locations. The input and desired output are encoded into the program via an expanded argv/stdin. In this encoding the desired output is assigned to `d` (not shown). String input and output is also possible by iterating over an output char array, checking it against the chars that would be created by stdout commands before they were transformed into these checks. Every assignment in the encoded C program is modified to allow the symbolic program verification tool to exhaustively explore different assignment values when that location is activated, creating the program model. In the example this is done with a toggle variable `t` which selects the location to explore with the symbolic call in Figure 1b. Finally, this model is "inverted" [10] which forces the verifier to suppress any assertion failures in the original model but generate new counter-examples if the program can terminate normally without violating these assertions. This is achieved by converting assertions to assumptions in the model and replacing any point where the program ends successfully with a new failing assertion, shown in Figure 1c.

This complete transformation effectively searches the model to find assignments which are possible repair locations. That is, every such assignment can be edited to a symbolic value which corrects the flow of *one* failing test case and results in the desired output being generated, i.e. not violating the `assume(o==d)` statement in the worked example's inverted model. Assignments where a symbolic value is found for *every* failing test case are returned as repair locations.

The localisation process effectively generates a look-up table at each returned location that will repair all failing test cases. For each failing test case, the alternative value of the assignments will be reported (via a counter-example) for each location flagged by this process. The flagging process means the chosen assignment values lead to the end of the program without failure of the original specification. All passing test cases define their own correct values for the assignments (they already execute within the test suite specification). So this look-up table is a genuine repair for the full test suite, albeit repairing only from the test cases provided as specification.

A multiprocess design that takes advantage of modern consumer processor architectures is used to accelerate the process [3]. Each test case is dispatched as a task to a worker pool with any pruning percolated to future tasks. The symbolic execution load is minimised by two features. Firstly, intelligent pruning of the search space, i.e. not searching known-unrepairable assignments in subsequent test cases after the search has started. Secondly, minimising the disruption of an intractable (a risk of model-checking programs) or slow search branch by monitoring and evicting tasks to be retried later after the search space has been pruned.

The end result of this method is that we generate a genuine list of repair locations as specified by test cases for any repair that could be expressed as a look-up table for the right-hand side of an assignment, within the limits of symbolic analyser accuracy.

5. EXPERIMENTS AND DISCUSSION

5.1 Experimental Setup

We generated all results on a 3.1GHz Core i5-2400. Our tool, using the Klee 1.1 symbolic analyser, operates as described in section 4. Spectrum-based fault localisation results using Ochiai and Tarantula formulas [37] are provided by Hawk-Eye [11]. As the rank of the fault location did not vary between formulas with this data set, both are referred to simply as the Hawk-Eye result.

5.2 Evaluation Strategy

Localisation tools typically produce a ranked list of locations, based on the generated suspiciousness value of each location. The rank is calculated using Hawk-Eye's middle-line strategy, ranking equally suspicious statements with the mid-point rank. For example, statements with suspiciousness (0.1, 0.4, 0.8, 0.4, 0.4) are ranked (5th, 3rd, 1st, 3rd, 3rd). Results from our model-based tool have been converted from an unranked list using the same middle-line strategy (where all locations returned are given a suspiciousness of 1.0 and all others a suspiciousness of 0.0).

Localisation performance is reported in the average percentage of other locations that will be searched before the fault is found when iterating over the locations in rank order. So in the above example, if the second statement is the fault location then the ranked localisation scores 50%. If the first statement (which is ranked last) was the fault then this would score 100%, the worst possible score, indicating every other returned statement would be searched before the fault was reached. A score near 0%, indicating very few locations ranked above the location used by the student to repair the submission in the database, means fewer instances of the debugging process stalling before repair synthesis can begin. Spectrum-based methods have been shown to provide rankings insufficient for novices [22] or even be unable to discriminate between most locations [24].

5.3 Results for Original Test Suites

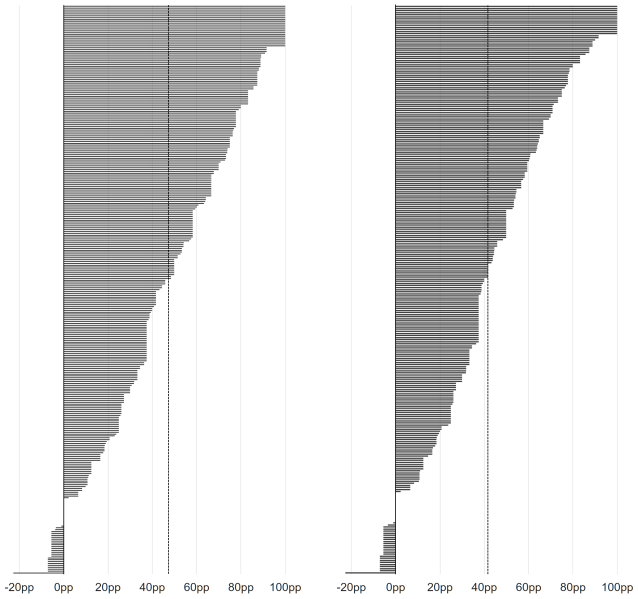
On the 304 failing student submissions and the instructor-authored test suites, our tool returns the faulty assignment after, on average, only 16% of other assignments. In Figure 2a the p.p. difference to the spectrum-based results (i.e. percentage points closer to a perfect 0% localisation) is plotted for every student submission, ordered by relative advantage. Positive differences indicate our tool's advantage. Hawk-Eye, on this data set, ranks the statement the student later used to repair the program after an average of 63% of other statements, 47 percentage points adrift (dashed line). No correlation was found between the program size and the comparative performance of our tool compared with Hawk-Eye.

On average, Hawk-Eye is provided eight test cases for each submission which are used to rank eleven statements. Our tool is provided with an average of four failing test cases, due to the small test suite size, and localises over the average of five assignments in each submission. These localisations are produced in an average of 0.3 seconds per submission by both our tool and Hawk-Eye. 254 times our tool was a fraction of a second ahead, 48 times Hawk-Eye was a fraction of a second ahead, and twice our tool hit too many pathological cases to adapt and only provided results at the tool's ten second time-out.

Our tool regularly pin-pointed the assignment later used by the student to bring the submission into compliance with the test suite. 125 of the 304 student submissions were returned from our tool with only that assignment flagged, the ideal result [22]. All other assignments were eliminated as viable repair candidates for the test suite specification. The limited number of failing test cases did not hinder our tool on this sample of short, real-world student programs.

When looking at those 125 submissions, Hawk-Eye ranked the repair statement after an average of 65% of other statements. Only 15 of those submissions provided the repair in the first third of the ranked list. The best ranking from Hawk-Eye for the statement the student used to repair the program was after 25% of other statements.

This comparison strongly supports the use of our tool for assisting students in repairing single-assignment faults in small program submissions, even when only specified by a very small test suite. When a student has made a mistake on constructing an assignment in an otherwise solid submission, an expensive debugging process may be averted by use of this feedback.



(a) Original Test Suite (b) Extended Test Suite

Figure 2: Comparison of Results: Our Tool vs Hawk-Eye

5.4 Results for Extended Test Suites

When the large test suite was used, Hawk-Eye showed a modest improvement. Although Figure 2b shows it continued to lag our tool significantly, at a 42 percentage point deficit (dashed line). Running an average of 154 test cases through each submission resulted in localisation of the repair statement to rank below 58% of other statements. Our tool, provided with an average of 80 failing test cases, did not improve from the 16% localisation score achieved previously. This lack of further improvement is due to a combination of different factors. Over a third of these submissions, with the smaller test suite, already provided perfect results using our tool; some programs will contain several assignments where a genuine repair is possible so there is no more compact list of repair locations; and some programs are resistant to analysis by symbolic analysis, which does not change with test suite size. This may have provided very little room for improvement by our tool.

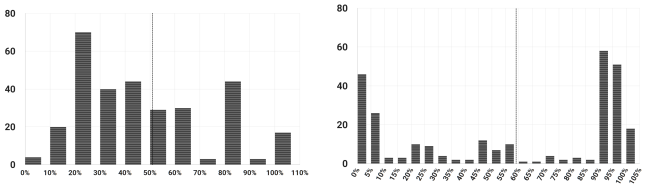
However, the run-time on these much larger test suites confirm the scalability of our tool, an area where model-based fault localisation has traditionally suffered [10]. Our tool averaged 0.9 seconds per submission while Hawk-Eye lagged behind, averaging 1.1 sec-

onds. Included in that average, three times our tool hit too many pathological cases to adapt and only completed at the time-out.

Hawk-Eye never ranked the statement that was used to repair the program as the most suspicious when localising on the data set, with either the original or extended test suite.

5.5 Effects of Programming Language

Comparing data for submissions originally written in Java to those in Python showed no significant skew to the data points explored. Translating a simple subset of each language into C syntax generated comparable subsets.



(a) Original Test Suite (b) Extended Test Suite

Figure 3: Histogram of Test Suite Failure Percentage

5.6 Grading Support

To confirm the value of this localisation data for detecting "almost correct" student submissions that test suites do not highlight, we extracted the test suite results for the data set. As each of these student submissions was selected because there exists a single edit to an assignment which brings it into compliance with the complete instructor-authored test suite, they should be graded within a generally narrow distribution, reasonably close to a fully compliant solution. However, as Figure 3a shows in a histogram of the test suite compliance score of these programs, this is not the case. The average submission fails for 51% of the test cases (dashed line) and the distribution of scores is very uneven, not clustering around a single value. As there are so few test cases per submission, the histogram buckets have been set to best show the distribution curve.

When expanding out to the much larger test suite in Figure 3b, the average submission fails for 59% of the test cases (dashed line). Here the distribution of grades is less flat, with clumping at both poles. Half of the "almost correct" submissions are scored with 80% or more of the test suite failing. Our tool will accelerate automation-assisted marking by flagging nearly-good code with the likely location of a repair that will radically increase a test suite-based grading.

5.7 Threats to Validity

Our tool and the slice used on the database of student programs to generate the pairs for analysis makes a single-fault assumption. This is a common assumption in the fault localisation field but does not reflect real world debugging, although the existence of many code pairs in this data set does confirm real-world applicability.

The slicing of the student programs via a simple language translation stage, without any translation of library calls, and rejection of unparseable code restricts the form of programs explored. This slice assumes a repair possible via assignment modification. The choice of symbolic analyser (with an integer solver) also restricts the type of programs processed. Some programs are not suitable for automated localisation, such as those that never terminate on some inputs, and these would not make it through the database slice. These restrictions could add a bias to the student programs explored which could unfairly advantage one tool or call the generalisability of these results into question.

The open-source script used to execute the spectrum-based localisation was written in Java. Executing Java scripts is known to come with a high initialisation time cost to start the JVM. Due to the short run-times involved, this may have inflated the run-time costs of this technique beyond some competing implementations based on the same GCov underlying tool.

6. CONCLUSIONS AND FUTURE WORK

Novice programmers are unlikely to be capable of advanced debugging techniques. Coursework and charettes provide an opportunity for fast, accurate fault localisation to assist in educational institutions, which have already built up databases and workflows around test suite specified exercises. We have demonstrated a fast, model-based fault localisation tool on a collection of single-fault, real-world student submissions. The high quality localisation information reduces the search space for novice debuggers working down a list of potential repair locations when compared to the results from spectrum-based techniques (cf. RQ2). In over a third of the sampled failing student programs, our tool used the test suite to provide a localisation result that uniquely identified the location later used by the student to repair the program and rejected all other locations. This reinforces the qualitative difference between model-based fault localisation that reasons over a model of the student program to derive a list of feasible repair locations compared to a spectrum-based ranking process that infers suspiciousness of each program statement (cf. RQ1). These short, often exact, lists of potential assignment repair locations can direct students to the site of improvement, assisting in the construction of a final submission that fully complies with the test suite.

Spectrum-based techniques showed marginal location ranking improvements with much larger test suites than found in the provided data set. This was, however, still vastly inferior to our tool's output (which did not significantly vary with test suite size) and came with an increased run-time cost. The run-time cost of our high quality localisation matches that of fast spectrum-based fault localisation, even with large test suites, ensuring that our approach is viable even at the scale required by MOOCs (cf. RQ1).

We have demonstrated that these submissions, which are a single assignment edit away from full compliance with a test suite, would not be scored predictably if scoring were based on compliance with the test suite only. This confirms the need for tools which can isolate such "almost correct" student submissions (cf. RQ3).

Future studies into this tool can survey the real world performance of tool-assisted learning in classroom environments and the effect on student progress when one group is provided with this high quality localisation information in typical student programming tasks. Such studies can validate this feedback as valuable for novices, improving total debugging time and reducing the number of students who stop before completing a source code submission fully conforming to the provided test suite. The integration of this localisation feedback into student integrated development environments must be managed to maximise and quantify student comfort and views on the ease-of-use of this additional information. Future studies could be done with direct interaction tracking, surveys, or analysis of achievement changes when this tool is introduced to an existing course.

The underlying methods used by this tool are programming language and symbolic analyser agnostic. This will allow future tests using other programming languages and beyond the restrictions of the currently selected downstream components.

7. ACKNOWLEDGEMENTS

We would like to thank Ewan Tempero and Paul Denny at the University of Auckland for access to an anonymised copy of their student exercises and submissions to those exercises, from which we generated our data set. We would also like to thank Eli Bendersky, author of the PyCParser; the Automated Software Testing Group at Beihang University for Hawk-Eye; all the contributors to the KLEE symbolic virtual machine project; and contributors to all downstream components of these tools and Linux Mint.

This academic research was funded by the Engineering and Physical Sciences Research Council, UK. Funding number 1239954.

8. REFERENCES

[1] P. Antonucci et al. An Incremental Hint System For Automated Programming Assignments. *ITiCSE '15*, pp. 320–325, 2015.

[2] S. D. Benford et al. The Ceilidh System for the Automatic Grading of Students on Programming Courses. *ACM Southeast Regional Conf.*, pp. 176–182, 1995.

[3] G. Birch, B. Fischer, and M. R. Poppleton. Fast Model-Based Fault Localisation with Test Suites. *TAP '15*, *LNCS 9154*, pp. 38–57, 2015.

[4] H. Blau and J. Eliot. FrenchPress Gives Students Automated Feedback on Java Program Flaws. *ITiCSE '15*, pp. 15–20, 2015.

[5] K. Buffardi and S. H. Edwards. Exploring Influences on Student Adherence to Test-Driven Development. *ITiCSE '12*, pp. 105–110, 2012.

[6] K. Buffardi and S. H. Edwards. Responses to Adaptive Feedback for Software Testing. *ITiCSE '14*, pp. 165–170, 2014.

[7] J. deKleer and B. Williams. Diagnosing Multiple Faults. *J. Artificial Intelligence*, 32(1):97–130, 1987.

[8] P. Denny, A. L. Reilly, and D. Carpenter. Enhancing Syntax Error Messages Appears Ineffectual. *ITiCSE '14*, pp. 273–278, 2014.

[9] S. H. Edwards. Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. *SIGCSE '04*, pp. 26–30, 2004.

[10] A. Griesmayer, S. Staber, and R. Bloem. Automated Fault Localization for C Programs. *Electronic Notes in Theoretical Computer Science*, pp. 95–111, 2007.

[11] Hawk-Eye: 2010. *http://code.google.com/p/hawk-eye/*.

[12] M. T. Helmick. Interface-based Programming Assignments and Automatic Grading of Java Programs. *ITiCSE '07*, pp. 63–67, 2007.

[13] D. Insa and J. Silva. Semi-Automatic Assessment of Unrestrained Java Code: A Library, a DSL, and a Workbench to Assess Exams and Exercises. *ITiCSE '15*, pp. 39–44, 2015.

[14] D. Jackson and M. Usher. Grading Student Programs Using ASSYST. *SIGCSE '97*, pp. 335–339, 1997.

[15] M. Vujošević-Janičić et al. Software Verification and Graph Similarity for Automated Evaluation of Students' Assignments. *Inf. Softw. Technol.*, 55(6):1004–1016, 2013.

[16] M. Joy, N. Griffiths, and R. Boyatt. The Boss Online Submission and Assessment System. *J. Educational Resources in Computing*, 5(3):2A, 2005.

[17] M-J. Laakso et al. Automatic Assessment of Exercises for Algorithms and Data Structures - a Case Study with TRAKLA2. In *Finnish/Baltic Sea Conf. on Comp. Sci. Edu.*, pp. 28–36, 2004.

[18] R. Lister et al. Naturally Occurring Data As Research Instrument: Analyzing Examination Responses to Study the Novice Programmer. *SIGCSE Bull.*, 41(4):156–173, 2010.

[19] T. MacWilliam and D. J. Malan. Streamlining Grading Toward Better Feedback. *ITiCSE '13*, pp. 147–152, 2013.

[20] L. Malmi, A. Korhonen, and R. Saikkonen. Experiences in Automatic Assessment on Mass Courses and Issues for Designing Virtual Courses. *ITiCSE '02*, pp. 55–59, 2002.

[21] M. McCracken et al. A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-year CS Students. *ITiCSE-WGR '01*, pp. 125–180, 2001.

[22] C. Parnin and A. Orso. Are Automated Debugging Techniques Actually Helping Programmers? *ISSTA '11*, pp. 199–209, 2011.

[23] D. N. Perkins et al. Conditions of Learning in Novice Programmers. *J. Educational Computing Research*, 2(1):37–55, 1986.

[24] L. H. Pham et al. Assisting Students in Finding Bugs and their Locations in Programming Solutions. *Int. J. Quality Ass. in Eng. and Tech. Edu.*, 3(2):12–27, 2014.

[25] V. Pieterse. Automated Assessment of Programming Assignments. *Comp. Sci. Edu. Research Conf.*, CSERC '13, pp. 45–56, 2013.

[26] A. L. Reilly et al. On the Differences Between Correct Student Solutions. *ITiCSE '13*, pp. 177–182, 2013.

[27] R. Reiter. A Theory of Diagnosis from First Principles. *J. Artificial Intelligence*, 32(1):57–95, 1987.

[28] R. Saikkonen, L. Malmi, and A. Korhonen. Fully Automatic Assessment of Programming Exercises. *ITiCSE '01*, pp. 133–136, 2001.

[29] R. Singh, S. Gulwani, and A. S. Lezama. Automated Feedback Generation for Introductory Programming Assignments. *PLDI '13*, pp. 15–26, 2013.

[30] E. Soloway and J. C. Spohrer. Studying the Novice Programmer. 1988.

[31] J. Spacco et al. Towards Improving Programming Habits to Create Better Computer Science Course Outcomes. *ITiCSE '15*, pp. 320–325, 2015.

[32] M. Striewe, M. Balz, and M. Goedicke. A Flexible and Modular Software Architecture for Computer Aided Assessments and Automated Marking. *CSEDU '09*, pp. 54–61, 2009.

[33] N. Tillmann et al. Teaching and Learning Programming and Software Engineering via Interactive Gaming. *ICSE '13*, pp. 1117–1126, 2013.

[34] Q. Wang, C. Parnin, and A. Orso. Evaluating the Usefulness of IR-Based Fault Localization Techniques. *ISSTA '15*, pp. 1–11, 2015.

[35] J. Whalley and N. Kasto. A Qualitative Think-Aloud Study of Novice Programmers' Code Writing Strategies. *ITiCSE '15*, pp. 320–325, 2015.

[36] D. Wood, J. S. Bruner, and G. Ross. The Role of Tutoring in Problem Solving. *J. Child Psychology and Psychiatry*, 17(2):89–100, 1976.

[37] X. Xie et al. A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization. In *TOSEM '13*, 22(4):31A, 2013.

[38] A. Zeller. Isolating Cause-Effect Chains with AskIgor. *IWPC '03*, pp. 296–297, 2003.