Using Property-Based Testing to Generate Feedback for C Programming Exercises

Pedro Vasconcelos
Computer Science Department, Faculty of Science, University of Porto, Portugal
LIACC, Porto, Portugal
pbv@dcc.fc.up.pt

Rita P. Ribeiro
Computer Science Department, Faculty of Science, University of Porto, Portugal
LIAAD-INESC TEC, Porto, Portugal
rpribeiro@dcc.fc.up.pt

Abstract

This paper reports on the use of property-based testing for providing feedback to C programming exercises. Test cases are generated automatically from properties specified in a test script; this not only makes it possible to conduct many tests (thus potentially find more mistakes), but also allows simplifying failed tests cases automatically.

We present some experimental validation gathered for an introductory C programming course during the fall semester of 2018 that show significant positive correlations between getting feedback during the semester and the student's results in the final exam. We also discuss some limitations regarding feedback for undefined behaviors in the C language.

2012 ACM Subject Classification Social and professional topics → Student assessment; Software and its engineering → Software testing and debugging; Software and its engineering → Domain specific languages

Keywords and phrases property-based testing, C language, Haskell language, teaching programming

Digital Object Identifier 10.4230/OASIcs.ICPEC.2020.28

1 Introduction

This paper reports on the use of property-based testing for automatically generating feedback to exercises in an introductory C programming course. As part of weekly worksheets, students were given assignments with automatic tests; they submitted solutions through a web-system and got feedback automatically. The tests assessed functional correctness, reporting the test data leading to a failure in case of wrong answers. Students could also perform multiple attempts, either to fix mistakes or to try alternative solutions.

Instead of fixed test suites, test cases are generated randomly from *properties* in a test script. This not only makes it easier to conduct more tests (thus potentially find more mistakes), but also allows reporting failed test cases: because new tests are generated, students cannot "cheat" simply by submitting solutions that are over-fitted to previously-reported cases. Furthermore, *shrinking* heuristics can be used to simplify failing cases automatically; this is helpful because randomly generated data often contains "noise" that is not relevant to the failure. Reporting shorter examples should help students debug their code and clarify misunderstandings.

This paper is structured as follows: Section 2 presents some related work; Sections 3 and 4 review property-based testing and describe our testing framework; Section 5 presents the experimental results; and Section 6 presents conclusions and directions for further work.

2 Related work

There has been increasing interest in the use of automated testing as an aid for teaching programming. Keuning et al. present a systematic comparison of tools for automatic feedback generation for programming exercises [10].

Gao et al. employ *concolic testing* to generate automatic feedback for a C programming course [8]; this approach is based on symbolic execution, and when compared to the one presented here, requires extra work to setup and process each exercise.

Fisher and Johnson describe the use of formal specifications as test case generators for Java programs [7]; this was done in the context of a software engineering course where understanding and writing specifications are part of the learning outcomes.

Our work is more closely-related to the one by Earle et al. [4] on using property-based testing to automatically assess Java programs. The main differences are: 1) we use Haskell instead of Scala for specifications; 2) we test C rather than Java programs and deal with the C specific issues such as undefined behaviours; and 3) we focus on functional correctness rather than grading.

3 Property-based testing

Property-based testing consists of specifying general assertions about software units and using these as test oracles; the actual test data can then be automatically generated (e.g. sampled randomly). The first property-based testing tool was the *QuickCheck* library for the Haskell language [5]. Similar libraries have since been developed for other languages, e.g. Erlang [1], Scala [12] and Python [11].

Properties are universally quantified assertions; testing these requires generating values and checking the assertion for each case. Passing a suitable large number of tests increases confidence in the correctness of the property. A single failed test, however, provides evidence that the property does not hold. Moreover, the failed test case can then be simplified automatically by applying "shrinking" heuristics for generated values.[1] Shrinking is useful because it can automatically remove irrelevant details from randomly generated data, leading to shorter and more insightful counter-examples.

Libraries for property-based testing provide functions for defining properties and generators from simpler ones and to control the number of tests and size data, making it possible to apply this methodology to real programs [3, 9].

4 Testing framework

4.1 Overview

Exercises consisted of individual C functions rather than complete programs. Alongside the problem description, the instructor sets up a test script written using *codex-quickcheck*[2], a custom version of the Haskell QuickCheck library with functionality to ease generating C

[1] For example, a strategy for shrinking strings is to try substrings of smaller length, or to replace some characters with strictly smaller ones.

[2] Available at `https://github.com/pbv/codex-quickcheck`.

types, testing C functions and producing student-friendly reports. This library is part of the *Codex* system for programming exercises[3] but can be used independently. The testing workflow consists of:

1. compiling the student code to an object file;
2. compiling and linking the test script together with the object file;
3. running the executable (under a sandbox) to produce an accepted/failure report.

Submissions are classified as follows:

CompileError rejected attempt due to a compiler error or warning;

RuntimeError, TimeLimitExceeded or MemoryLimitExceeded the execution was aborted due to a runtime error or resource exhaustion;

WrongAnswer testing found a witness that falsifies a property;

Accepted all tests passed.

Because the exercises were not computationally intensive, the turn-around for student feedback was quick (typically 2-4 seconds). Since students could edit code directly on the web interface and there was no penalty for repeated submissions, many submissions were made even for short assignments.

4.2 Example: testing the median function

Listing 1 Test script for the median function exercise.

```
1  import Data.List (sort)
2
3  foreign import ccall "median" median :: CInt -> CInt -> CInt -> CInt
4
5  prop_correct
6    = testing "median" $
7      forArbitrary "a" $ \a ->
8      forArbitrary "b" $ \b ->
9      forArbitrary "c" $ \c ->
10         median a b c ?== sort [a,b,c] !! 1
11
12 main = quickCheckMain prop_correct
```

Listing 1 presents the script for testing a sample exercise: *write a function to compute the median of three numbers.* Line 3 imports the C function to be tested using the standard Haskell foreign-function interface. Lines 5–10 specify the correctness property: the median of a, b, c is the value at index position 1 (i.e. the middle element) of the sorted list of values.[4] The `forArbitrary` function introduces a quantified property using the default generator (for C integers, in this case) and also names the variable for reporting. The assertion operator `?==` compares the left-hand side with the (expected) right-hand side, producing a suitable message otherwise.

Listing 2 shows a hypothetical feedback for a submission that gives a wrong result when two arguments are equal. Reports always follow this format: the name of the function being tested, the names and values of arguments and the expected and obtained results. Students can use reports to clarify their understanding of the exercise, or as a starting point for debugging, without any knowledge of Haskell or property-based testing.

[3] Available at `https://github.com/pbv/codex`.
[4] `!!` is the list indexing operator in Haskell.

Listing 2 Example feedback for the median function exercise.

```
*** Failed! Falsified (after 1 test and 2 shrinks):
Testing median with:
        a = 0
        b = 0
        c = 1
Expected:
        0
Got:
        1
```

In principle, properties are quantified over all possible values; in practice, they are tested with a finite sample of randomly-generated test cases (100 by default). The number of tests and the maximum size of values are configured as metadata for exercises; it is also possible to fix the seed for pseudo random-number generation (and thus get reproducible results). Test cases are generated with increasing sizes, so that the smaller examples are tried first. Furthermore, shrinking simplifies failed examples before reporting.

4.3 Custom generators and shrinking

The previous example used the default generators for integers. In general, we need to be able to define properties using custom generators. Listing 3 presents a test script for such an example, namely, a function that checks if a string is a "strong password" according to the following rules: it should have at least 6 characters and contain one lowercase, one uppercase and one digit character.

Lines 6–8 define a functional wrapper for the C function to be tested; it uses `withCString` to convert a Haskell string into a C character buffer[5], ensuring proper deallocation. Lines 10–13 define a Haskell specification for the solution; lines 15–18 specify the correctness property using `forAllShrink` instead of `forArbitrary`; this allows using a *custom generator* (line 20) and *shrinking function* (line 21) that generates strings containing only a subset of printable ASCII characters. The generator is defined using functions `listOf` and `choose` from the QuickCheck library; we use `shrinkMap` to apply the default shrinking for strings filtering out characters outside the desired range.

Listing 3 Test script for the strong password exercise.

```
1  import Data.Char(isUpper, isLower, isDigit)
2
3  foreign import ccall "strong_passwd"
4    strong_passwd :: CString -> IO CInt
5
6  strong_passwd_wrapper :: String -> CInt
7  strong_passwd_wrapper str
8    = runC $ withCString str strong_passwd
9
10 strong_spec :: String -> Bool
11 strong_spec str
12   = length str>=6 && any isUpper str &&
13     any isLower str && any isDigit str
14
```

[5] Haskell strings are linked-lists rather than contiguous buffers.

```
prop_correct
  = testing "strong_passwd" $
    forAllShrink "str" genPasswd shrinkPasswd $ \str ->
       strong_passwd_wrapper str ?== fromBool (strong_spec str)

genPasswd = listOf (choose ('0', 'z'))
shrinkPasswd = shrinkMap (filter (\c -> c>='0' && c<='z')) id

main = quickCheckMain prop_correct
```

Listing 4 presents reports based on a real student attempt that incorrectly assumed that all characters in the string must be letters or digits. The top report illustrates a random test case obtained (before shrinking), while the bottom one is obtained after shrinking; we report the later to students. This example illustrates the effectiveness of shrinking to automatically produce more insightful counter-examples.[6]

Listing 4 Example feedback for the strong password function exercise with shrinking disabled and enabled.

```
*** Failed! Falsified (after 16 tests):
Testing strong_passwd with:
        str = "gSvF<NiXz]BH_"
Expected:
        0
Got:
        1

*** Failed! Falsified (after 16 tests and 7 shrinks):
Testing strong_passwd with:
        str = "aaaaA_"
Expected:
        0
Got:
        1
```

4.4 Mitigating undefined behavior

One of the challenges of teaching the C language is the need to alert students to avoid inadvertently causing *undefined behaviors* (UB). It is important to catch UB when doing automated testing because they can lead to puzzling results (e.g. non-deterministic answers across operating systems, compiler versions, etc.). While ensuring the complete absence of UB is quite difficult, we employed some simple mitigation techniques using the GNU C Compiler:

- *enabling exhaustive warnings* and treating them as compile-time errors;
- *enabling optimizations* (e.g. `-O1`) also enables static checks about potential errors (e.g. uses of uninitialized variables);
- *enabling the UB sanitizer* (e.g. `-fsanitize=undefined`) for introducing runtime checks for some UB (e.g. division by zero and integer overflows).

[6] The string found is, in fact, a *local minimum* that distinguishes the student's attempt from a correct solution: it has length 6, at least 1 lower and uppercase letter, but no digit.

Another mitigation technique is to perform all memory management and array initialization from the Haskell side (e.g. the `withCString` function). The *codex-quickcheck* library provides a *checked* initialization function that places random "canaries" [6] outside the array boundaries and checks for overwrites. This allows detecting and reporting off-by-one index errors in student code that performs array writes; however, it is not so effective at catching index errors for array reads (see also the discussion in Section 5.2 regarding exercise ex8_2).[7]

4.5 Developing specifications

We developed test scripts for 15 exercises covering elementary programming concepts (e.g. function definitions, conditionals, loops, arrays and strings). Most test scripts are short (e.g. under 50 lines) and required between 15 to 60 minutes to develop. Moreover, properties and generators developed for one exercise can often be easily adapted to related ones. The largest script has 116 lines (exercise ex9_5 in Table 1); this is due to code for generators of matrices that are "magic squares" and ones that fail each of the necessary conditions.

In order to check adequate test case distribution, first we developed the correctness properties alone, and subsequently collect statistics to adjust the testing parameters and generators. For the example of Section 4.3, we can count the percentage of "small" test cases (i.e. length less than 6) by introducing into line 18 of Listing 3:

```
classify (length str < 6) "small" $ ...
```

To also count which combinations of conditions are tested, we further add:

```
collect (any isUpper str, any isLower str, any isDigit str) $ ...
```

Running the test script with a reference solution we obtain:

```
+++ OK, passed 100 tests (20% small)
81% (True,True,True)
 9% (True,True,False)
 5% (False,False,False)
 2% (False,True,False)
 1% (False,False,True)
 1% (False,True,True)
 1% (True,False,False)
```

This shows that 20% of the generated strings were small and 81% satisfied all conditions; also, some combinations of conditions were poorly tested and one combination was untested. We can now improve this distribution by changing the test data generation; for this example it suffices to reduce the maximum data size (i.e. the maximum string length) and increase the number of tests.

5 Experimental evaluation

Throughout the semester students could submit exercise solutions using the web system; they were not penalized for multiple attempts and could also continue submitting even after having an accepted solution (e.g. to experiment with alternatives). Submissions were used as formative rather than summative assessment; however, a minimal number of correct submissions was required to qualify for the final exam.

[7] The GCC address sanitizer (`-fsanitize=address`) cannot help here because it does not track heap overflows in memory managed by the Haskell runtime system.

Table 1 Summary of exercises, total number of attempts, percentage of wrong answers and median (maximum) number of attempts per student.

Exercise	Description	#Attempts	%WrongAns	#Attempts per student
ex3_8	median of 3 integers	1024	17.2%	5.0 (41)
ex3_9	compute integer powers	1288	28.3%	3.5 (84)
ex4_5	sum integer divisors	740	30.4%	3.0 (28)
ex4_7	determine next leap year	1048	23.5%	4.0 (48)
ex5_8	approximate a power series	1103	45.6%	4.0 (67)
ex6_4	initialize an array	484	14.0%	2.0 (36)
ex6_8	test repeated values in array	712	38.2%	2.0 (33)
ex7_5	check strong passwords	767	27.8%	3.5 (24)
ex7_10	filter positive values in array	656	42.5%	1.5 (54)
ex8_2	check if an array is ordered	830	50.1%	3.0 (62)
ex8_7	insertion into an ordered array	1192	40.4%	4.0 (63)
ex9_2	check if 2 strings are anagrams	848	32.1%	3.0 (33)
ex9_5	check magic squares	785	47.3%	2.0 (74)
ex10_4	array max and min using pointers	539	42.3%	2.0 (25)
ex10_9	find character in a string using pointers	368	12.0%	1.0 (21)

Ideally, we would assess the effect of automatic feedback by comparing the learning results of students in two groups, one group getting the automatic feedback while the control group getting no feedback. However, this would be undesirable due to the differentiate treatment of students in the class. Instead, we followed an approach similar to Ahadi et al. [2] and performed a correlation analysis between students' submissions and their final results. In particular, we measured correlations between the *total number of attempts* and the *number of wrong answers* on each exercise and the students' final exam scores.

5.1 Experimental setup

The exam was attended by 152 students which, during the semester, had performed a total of 12384 submissions for the 15 proposed exercises. From these submissions, we gathered, for each student and each exercise: the number of attempts made (#Attempts); and the number of attempts classified with *Wrong Answer* (#WrongAns).

The data set consists of 2280 (152×15) observations. Table 1 lists, for each exercise, the total number of submissions made, the percentage of *Wrong Answers*, the median and the maximum number of attempts per student. Observe that some exercises have quite more attempts than others. Moreover, the percentage of *Wrong Answer* varies considerably. Note also that the median number of attempts per student is much smaller than the maximum, which indicates that only a few students perform such large amount of attempts.

For each of the 15 exercises, we classified students according to the following scenarios:

1. the student made $< 2^n$ attempts;
2. the student made $\geq 2^n$ attempts;
3. the student made $< 2^n$ *Wrong Answer* attempts;
4. the student made $\geq 2^n$ *Wrong Answer* attempts.

The maximum number of attempts per student for any exercise was 84 (cf. Table 1); thus, we have considered n ranging from 0 to 6 in the above scenarios.

We performed a *Pearson correlation test* for each of the binary variables generated in the 4 scenarios above and the binary variable indicating whether the student scored above the overall median grade of the exam. As all the variables involved in the correlation analysis are binary, this corresponds to obtaining the *phi* correlation coefficient. This coefficient is obtained on the basis of a contingency table for two binary variables, as exemplified in the Table 2. This table contains the number of students satisfying each of the four possible combinations for the two binary variables under analysis. One of the variables refers to one of the 4 scenarios for a given exercise and the other refers to the obtained score w.r.t. to the overall median score of the final exam. This means that each of the 15 exercises was tested against the final exam score, in all the possible scenarios.

Table 2 Contingency table for exercise X in the context of scenario Y w.r.t the overall final exam score median.

		final exam score ≥ median?	
		yes	no
exercise X meets criterion of scenario Y?	yes	*a*	*b*
	no	*c*	*d*

Based on Table 2, the *phi* coefficient is obtained by the following formula:

$$phi = \frac{ad - bc}{\sqrt{(a+b)(c+d)(a+c)(b+d)}} \qquad (1)$$

The correlation tests were performed using the function `cor` from the R statistical language [13]. The following criteria were used for pruning the contingency tables (similar to the ones used in [2]):

1. to avoid over-fitting [14], contingency tables with any cell value less than 5 were pruned;
2. contingency tables with negative *phi* were pruned; for each of these there is a "mirror image" table for the reversed criteria with a positive *phi* value;
3. if two contingency tables have *phi* values that differ less than 0.01 and one is more general than the other, the less general one was pruned (e.g. "≥ 2 attempts" is more general than "≥ 4 attempts");
4. if two contingency tables have *phi* values that differ more than 0.01 and the more general one has a higher *phi* value, the less general one was pruned.

5.2 Main results and discussion

Table 3 presents the results obtained by our tests that are significant with a 95% confidence level, i.e. with a p-value < 0.05.

One might expect that solving exercises in fewer attempts would correlate with better final results, but we detected more correlations with "≥" criteria; this was also observed in the previous work [2] and we conjecture that similar explanations apply here:

- some exercises with a high number of total attempts also exhibit high correlation with final results (e.g. ex5_8, ex8_7); perhaps these exercises trigger difficulties that students need to experience in order to improve their understanding;
- students could submit attempts out of class, thus if they performed too few submissions it could simply indicate they were getting solutions from someone else;
- finally, a high number of attempts could also be an indicator that the student used the system specifically as a study aid for the exam.

Table 3 Significant correlations between the number of attempts or wrong answers by exercise and the final exam score (p-value < 0.05).

Exercise	#Attempts	#WrongAns	*phi*	*p*-value
ex3_9	*	≥ 1	0.238	0.003
	≥ 2	*	0.190	0.019
ex4_5	*	≥ 2	0.179	0.027
ex5_8	*	≥ 1	0.277	0.001
	≥ 4	*	0.198	0.015
ex6_8	≥ 2	*	0.199	0.014
ex7_5	≥ 1	*	0.229	0.005
ex7_10	*	≥ 1	0.302	0.000
	≥ 1	*	0.272	0.001
ex8_2	*	< 4	0.196	0.016
	< 8	*	0.199	0.014
ex8_7	*	≥ 4	0.239	0.003
	≥ 1	*	0.287	0.000
ex9_2	*	≥ 1	0.267	0.001
	≥ 1	*	0.272	0.001
ex9_5	*	≥ 2	0.303	0.000
	≥ 2	*	0.254	0.002
ex10_4	*	≥ 1	0.174	0.032
	≥ 1	*	0.230	0.004
ex10_9	≥ 1	*	0.200	0.013

The results for exercise ex8_2 show an unusual pattern: criteria with “$<$” conditions correlated better with final results, while criteria with “$\geq$” do *not* exhibit significant correlation. Also, this exercise had an unusually high percentage of *Wrong Answer* outcomes (cf. Table 1). Analyzing the submissions, we observed that a large number of *Wrong Answers* were actually caused by reading past the end of the array. The testing library does not detect this error as an undefined behavior (see Section 4.4) and the test case reported might not be easily reproducible; this might explain why the feedback obtained was less helpful.

For the remaining exercises in Table 3 having a higher number of attempts correlated positively with better student’s results, and in particular, for exercises ex3_9 ex4_5, ex5_8, ex7_10 and ex9_5, the number of *Wrong Answer* attempts had a higher and more significant correlation than the number of attempts alone. While these results do not prove causality, they are consistent with the hypothesis that automatically generating feedback has improved students’ results.

6 Conclusion and further work

We have presented the use of property-based testing for providing automatic feedback to C programming exercises. Our approach is to use a library in a high-level language for defining properties to generate many random test cases. Furthermore, the test cases can be automatically simplified before reporting to students. Experimental results indicate that there were significant positive correlations between the use of this testing system and the students’ final results.

One direction for further work is to integrate an interactive simulator into the feedback loop. Automatically invoking a web-based system such as the *C Tutor*[8] should help students investigate failures (we recommended students performed this step manually in classes). The visualizer could also help students because it explicitly highlights undefined behaviors as errors.

To test interfaces with multiple functions rather than single ones we could use the finite-state machine approach of the Erlang QuickCheck [9]; this could be useful for more advanced C programming courses (e.g. on data structures or operating systems).

References

1 QuviQ AB. Erlang QuickCheck. `http://www.quviq.com/products/erlang-quickcheck/`, 2018. [Online; accessed April 2020].

2 Alireza Ahadi, Raymond Lister, and Arto Vihavainen. On the number of attempts students made on some online programming exercises during semester and their subsequent performance on final exam questions. In *Proc. of the 2016 ACM Conf. on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 218–223. ACM, 2016.

3 Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with Quviq QuickCheck. In *Proc. of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, pages 2–10. ACM, 2006.

4 Clara Benac Earle, Lars-Åke Fredlund, and John Hughes. Automatic grading of programming exercises using property-based testing. In *Proc. of the 2016 ACM Conf. on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 47–52. ACM, 2016.

5 Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proc. of the Fifth ACM SIGPLAN International Conf. on Functional Programming*, ICFP '00, pages 268–279. ACM, 2000.

6 Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. *Foundations of Intrusion Tolerant Systems*, pages 227–237, 2003.

7 Gene Fisher and Corrigan Johnson. Making formal methods more relevant to software engineering students via automated test generation. In *Proc. of the 2016 ACM Conf. on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 224–229. ACM, 2016.

8 Jianxiong Gao, Bei Pang, and Steven S. Lumetta. Automated feedback framework for introductory programming courses. In *Proc. of the 2016 ACM Conf. on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 53–58. ACM, 2016.

9 John Hughes. *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*, pages 169–186. Springer International Publishing, 2016.

10 Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. Towards a systematic review of automated feedback generation for programming exercises. In *Proc. of the 2016 ACM Conf. on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 41–46. ACM, 2016.

11 David R. MacIver. Hypothesis. `https://hypothesis.readthedocs.io/`, 2018. [Online; accessed April 2020].

12 Rickard Nilsson. Scalacheck: Property-based testing for Scala. `https://www.scalacheck.org/`, 2018. [Online; accessed April 2020].

13 R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, 2019. URL: `https://www.R-project.org/`.

14 Frank Yates. Contingency tables involving small number and the χ^2 test. *Supplement to the Journal of the Royal Statistical Society*, 1(2):217–235, 1934.

[8] `http://www.pythontutor.com/c.html`