Prof. Dr. A. Podelski,
Dr. B. Westphal
M. Steinle

Summer term 2020

Softwaretechnik/Software Engineering

http://swt.informatik.uni-freiburg.de/teaching/SS2020/swtvl

Exercise Sheet 5

Early submission: Monday, 2020-07-13, 14:00 Regular submission: Tuesday, 2020-07-14, 14:00

Please note the extended deadline for the Bonus Exercise 5 (see below).

Exercise 1 – Testing & Coverage Measures (10/20 Points)

```
1  int convert(char[] str) throws Exception {
2          if (str.length > 6)
3                  throw new Exception("Length exceeded");
4          int number = 0;
5          int digit;
6          int i = 0;
7          if (str.length > 0 && str[0] == '-')
8                  i = 1;
9          while (i < str.length){
10                 digit = str[i] - '0';
11                 if (digit <= 0 || digit > 9)
12                         throw new Exception("Invalid character");
13                 number = number * 10 + digit;
14                 i = i + 1;
15         }
16         if (str.length > 0 && str[0] == '-')
17                 number = -number;
18         if (number > 32767 || number < -32768)
19                 throw new Exception("Range exceeded");
20         return number;
21 }
```

Figure 1: Function `convert`.

Consider the Java function `convert` shown in Figure 1. The function is supposed to convert the (up to 6-digit) string representation (decimal, base 10) of a 16-bit number to the represented integer. The following exceptional cases should be considered, i.e. corresponding exceptions should be thrown:

- The input string `str` has strictly more than 6 characters.
- The input string has at most 6 characters, and one of them is not a digit, i.e. from {'0', ..., '9'} (the first character may in addition be a minus ('-')).
- The input string has at most 6 characters, all of them digits (possibly a '-' in front) and the denoted integer value is outside the range $[-32768, 32767]$.

If none of the exceptional cases applies, let $c_0, \ldots, c_{n-1}$, $n \geq 0$, be the first, second, ..., n-th character in `str` (from left to right). The expected return value is $\sum_{i=0}^{n-1} c_i \cdot 10^{n-i-1}$ if the first character c_0 is not '-' and $-\sum_{i=1}^{n-1} c_i \cdot 10^{n-i-1}$ if the first character c_0 is '-'. (Note that, unlike other implementations of string-to-integer conversions, this one should return 0 for the empty string and the string "-".)

(i) Give an *unsuccessful* test suite for `convert` that achieves 100% *statement coverage* and 100% *branch coverage*. (6)

Hint: Make sure that your presentation easily and strongly convinces your tutor about your claims on the test cases and the coverage measures.

(ii) Modify your test suite from Task (i) such that the test suite is still *unsuccessful* and still achieves 100% *statement coverage*, but now achieves *strictly less* than 100% *branch coverage*. (1)

(iii) Is `convert` correct wrt. the (functional) specification detailed above? Argue your claim. (1)

(iv) What is the number of test cases required to *exhaustively* test `convert` for only the strings of length six? Assume that a character in a string has size 16-bit. (1)

Extra exercise without points: How many test cases are needed for all strings of length up to and including 7?

(v) If we can conduct around 1000 tests per second (say, 1024, for simplicity), how many seconds (hours? days? years?) would it then at least take to exhaustively test the program on only the strings of length six? (1)

Extra exercise without points: How long for all strings of length up to and including 7?

Exercise 2 – Testing in the Software Development Process (3/20 Points)

(i) What is, in general, the relation between the existence of an *unsuccessful* test suite with full (100 %) or partial (strictly less than 100 %) statement coverage for a function and correctness wrt. a given specification? (2)

Hint: In other words, we have to consider four cases that we could call correct-and-100 %-coverage, correct-and-less-than-100 %, incorrect-and-100 %, incorrect-and-less-than-100 % (in all cases, the test suite is assumed to be unsuccessful).

Are all four cases principally possible, or does, e.g., one correctness case imply a coverage case?

(ii) Assume that there is an unsuccessful test suite for a function and after releasing the software including this function the client or a user reports an incorrectness (a "bug"). That is, client or user do not just claim an incorrectness, but the incorrectness is confirmed by the developers and eliminated for future releases.

It is generally recommended to extend the test suite with at least one test case with inputs that demonstrate the incorrectness. What could be rationales behind this recommendation? (1)

Exercise 3 – Grading Test Suites (1/20 Points)

Assume that there is a task to create an unsuccessful test suite for the function shown in Figure 2(a) (a complicated implementation of the function $\max(0, n)$). The test suite should achieve 100% branch coverage and 100% statement coverage, and document how this coverage is achieved. Consider the proposed solution shown in Figure 2(b) (Figure 2(b) is the complete submission, no further text or explanation). Branches and statements are denoted by i_2, s_3, s_5 using the line number.

Assume that a correct solution would be worth 4 points. There is a proposal to grade the proposed solution shown in Figure 2(b) with 0 points. Argue in favour of this proposal.

```
1 int max0( int n ) {
2   if (n < 0)
3     return 0;
4   else
5     return n;
6 }
```

(a) Function `max0()`.

input	i_2^t	s_3	i_2^f	s_5
-1	✔	✔	✘	✘
27	✘	✘	✔	✔

(b) Proposed solution.

Figure 2: Function under test and proposed test suite.

Exercise 4 – Verification with PD Calculus (6/20 Points)

Consider the program `multiply` shown in Figure 3. It is supposed to implement multiplication of two non-negative integers by successive addition as specified by the given pre- and post-conditions. The operands are y and x and the result is stored in the variable r.

$\{y \geq 0 \wedge x \geq 0\}$

```
r = 0;
i = 0;
while (i < x) do
  r = r + y;
  i = i + 1;
od
```

$\{r = y \cdot x\}$

Figure 3: Program `multiply`.

The goal of this exercise is to use the proof system PD to show that `multiply` is partially correct. We approach this goal in three steps:

(i) Give a *loop invariant* for the while loop that enables you to prove the correctness of the program. (1)

Hint: If you do not have a good idea for a loop invariant, do not hesitate to ask your tutor to propose one so that you can continue with the other tasks.

(ii) Apply the rules of the proof system PD to *derive a proof* that your invariant is indeed a loop invariant (in other words: establish the premise of Rule 5).

Specify which rules or axioms you use at every proof step. (4)

(iii) Apply the rules of the proof system PD to *derive a proof* that `multiply` is *partially correct*.

Specify which rules or axioms you use at every proof step. (1)

Exercise 5 – Verification with VCC - Extended Deadline 2020-07-21, 14:00 (10 Bonus)

Note that feedback for this extended-deadline exercise will come together with the feedback on Exercise Sheet 6 even if you include your solution into your submission on this exercise sheet. In other words: If you choose to go for the bonus points, you may submit your solution together with your solutions on this Exercise Sheet 5 or together with your solutions on Exercise Sheet 6; in any case, feedback will come as scheduled for the latter.

(i) We implemented the program `multiply` from Exercise 4 as a C function (see the file `multiply.c` in Ilias). Assume that `multiply` is used in a larger program where it is only called with values for y between 0 and 15 and values for x between 0 and y, i.e., in the considered larger program, all callers guarantee the pre-condition $\{0 \leq x \leq y \leq 15\}$.

Annotate the function with pre- and post-condition, and a loop invariant (and whatever else you consider necessary) using the VCC syntax for annotations, and use the VCC tool[1] to see whether it is able to *verify* that the annotated function is correct wrt. pre- and post-condition. (3 Bonus)

(ii) Assuming that a solution for Exercise 4.(iii) exists, what should we have expected for the outcome of Exercise 5.(i)? Argue your expectation(s). (1 Bonus)

(iii) We have observed that with testing, it is easy to provide an *unsuccessful* test suite for a program which is *not correct* wrt. its specification. How is it with VCC? (2 Bonus)

Hint: As an experiment, change the C function such that the function is not *correct wrt. the specification anymore. Describe your modification (in how far it causes a violation of the specification?) and the behaviour of VCC when applied to the modified function.*

(iv) For the values $N = 15, 150, 1500, 15000$,

a) how many test cases are needed to exhaustively test `multiply` wrt. the pre-condition $\{0 \leq y \leq x \leq N\}$? (2 Bonus)

Hint: We need a term to compute the number of test cases for any given N, and (at least the orders of magnitude of the) particular numbers for the values of N given above.

b) If we use VCC to verify the function with the new pre-conditions (for the respective values of N), we observe that the verification time reported by VCC is approximately the same for all N. Is this measurement plausible? Why (not)? (1 Bonus)

(v) Assume that we were unsure about our proof from Exercise 4.(iii) and would like to confirm the result using VCC.

Modify the C function such that it in particular considers the original pre-condition (cf. Figure 3), and try to verify it using VCC. Describe what you expect to be the outcome of this experiment and what the results of the verification attempt with VCC were. Interpret the output of VCC. (1 Bonus)

To enable your tutor to reproduce your results, all code artefacts of this exercise (with appropriate comments or explanations, if necessary) should be submitted as separate text files, i.e., not embedded in the PDF.

[1] `http://rise4fun.com/vcc`