Prof. Dr. A. Podelski,
Dr. B. Westphal Summer term 2019

Softwaretechnik/Software Engineering

`http://swt.informatik.uni-freiburg.de/teaching/SS2019/swtvl`

Exercise Sheet 6

Early submission: Wednesday, 2019-07-24, 12:00 Regular submission: Thursday, 2019-07-25, 12:00

Exercise 1 – Black-box Testing (7/20 Points + Bonus)

Note: You need to contact your tutor to work on this exercise, as described in the appendix.

Assume that the customer has approved the Decision Table from Tutorial 2, Slide 27, as valid for the book discount calculation. Afterwards, a company has been sub-contracted to implement the book discount calculation as specified by the table in a program called `discount_calc`. The purpose of this program is, given the relevant data of a client and the base price of a book, to find out whether the client is allowed to borrow the book, and, if so, to determine the discounted price. The program `discount_calc` takes five input arguments (in this order):

- `O` is the number of books owned by the client.
- `B` is the number of books borrowed by the client.
- `R` is the number of bad ratings of the client.
- `S` is a flag for the student status of the client.
- `P` is the base price of the new book to be borrowed in cents.

`O`, `B`, `R`, and `P` are integers in the range [0, 32767], `S` is Boolean (i.e., an integer in the range [0, 1]). The program is supposed to yield the final price (with the discount already applied) in cents, and rounding is performed toward zero. Lending refusal is encoded by result value -1 and an invalid input by -2, thus the program is supposed to yield values in the range $[-2, 32767]$.

Users of the program observed some calculations being performed incorrectly. Unfortunately, they cannot remember which inputs caused the program to misbehave.

You are now in the role of the test engineer. You find out that the sub-contracted company refuses to look into the issue until you give evidence that the program does not correctly implement the table. Even worse, you understand that the contract does not require the sub-contracted company to hand over the source code to you and all records on the acceptance tests (conducted by your predecessor on the job) have (of course...) been lost. So you have to rely on pure black-box testing.

(i) Develop a *test suite* to test whether the program violates its specification.

Describe the strategy you used to select the test cases, and explain how this strategy is reflected in the test suite. For one of your test cases, discuss what exactly you can conclude from a positive and negative outcome of an execution of this test case, respectively. (5)

Submit a test suite via a test script according to the instructions in Appendix A. The test script must be submitted as a separate text file, i.e., not only embedded in the PDF with your other solutions.

Hint: You can assume that your boss allotted a time budget for the creation of the test suite equivalent to 5 exercise sheet points. Submit a number of test cases that is appropriate to this time budget.

Your tutor will execute your test suite (using your test script, hence it is important that you follow the file format given in the appendix). There are the following possibilities:

```
int convert(char[] str) throws Exception {
        if (str.length == 0)
                return 0;
        if (str.length > 6)
                throw new Exception("Length exceeded");
        int number = 0, digit, i = 0;
        while (i < str.length) {
                digit = str[i] - '0';
                if (digit <= 0 || digit > 9)
                        throw new Exception("Invalid character");
                number = number * 10 + digit;
                i = i + 1;
        }
        if (number > 65535)
                number = 65535;
        return number;
}
```

Figure 1: Function `convert`. Specification: For string $str = c_1, \ldots, c_m$, $m \geq 0$, if $m = 0$, yield 0; if $m > 6$, throw exception 'length exceeded'; if $m \leq 6$ and any of $c_1, \ldots, c_m$ is not a decimal digit (0, ..., 9) throw exception 'invalid character'; otherwise yield $\min(\sum_{i=1}^{m} c_i \cdot 10^{m-i-1}, 2^{16} - 1)$.

- Your test suite finds one error in the program. (1 Bonus)
- Your test suite finds a second error in the program. (3 Bonus)
- Your test suite finds a third error in the program. (5 Bonus)
- Your test suite finds a fourth error in the program. (100 Bonus)
- Your test suite finds more errors in the program. (1000 Bonus each)

Hint: For the assignment of bonus points, an error is a wrongly implemented rule, or a violation of an additional specification for the program as given above, like the encoding of Boolean flags. That is, your test suite may contain more than one successful test case, but each of them might identify the same error.

(ii) What is the number of test cases required to *exhaustively* test the program *inside its specification*? (1)

(iii) If we can execute around 1000 tests per second, how many seconds (hours? days? ...) would it then at least take to exhaustively test the program? (1)

Exercise 2 – Testing & Coverage Measures (7/20 Points)

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 exact specification (inclusive of the treatment of corner-cases) is given in the caption of Figure 1.

(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)

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 (5/20 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.
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)

(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. (3)

(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)

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

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

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

Figure 3: Program `multiply`.

Exercise 5 – Verification with VCC (10 Bonus)

(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`

A Instructions for the Black-box Testing Exercise

Before you begin, you need to request a team ID from your tutor by e-mail. We have created a separate binary to test for each team. Your tutor will assign a number from 0 to 99. The team ID will be used to evaluate your results on the particular implementation of the discount calculation. In order to create a test script, remote-login (via ssh(1)) to `login.informatik.uni-freiburg.de` and perform the following steps:

- Create a directory where you would like to store your results.
- In your directory of choice, execute the installation script as follows[2]:

  ```
  /home/westphal/testing/install.sh ID
  ```

  where `ID` is the team ID that you received from your tutor. The script will create the file `testsuite.sh` on the current directory.
- Use your favourite editor to open that file and insert one line per test case at the end of the file using the following format:

  `do_test` $\langle val_0 \rangle \; \ldots \langle val_{n-1} \rangle \; \langle val_n \rangle$

  where $\langle val_i \rangle$, $0 \leq i < n$, $n \in \mathbb{N}_0$, is the vector of *input* values of your test case and $\langle val_n \rangle$ (the last value in the list) is the *expected* (or: Soll) value.
- Please use only the characters 0–9, a–z, A–Z, and '-' to construct the values $\langle val_i \rangle$ in your submissions; test cases using other characters are not eligible for bonus points.
- You may document your test suite by using *dedicated lines* starting with '`#`' anywhere among your test cases; that is, character '`#`' on a `do_test`-line will count as part of your test case (which would harm eligibility for bonus points (see above)).
- Save the file and submit it with your other exercise solutions.

Running the tests (optional)

Note that Task (i) only requires you to submit your test script. However, if you would like to execute the test script yourself, the binaries for testing are accessible. You may execute them at your discretion. The script will execute your test cases on your team's implementation and report the results.

To run the test script, execute the command

```
./testsuite.sh
```

on the host

```
login.informatik.uni-freiburg.de
```

[2]Note that you cannot use the command-line completion feature; just enter the command as given above.