Prof. Dr. A. Podelski, Dr. B. Westphal
S. Feo Arenis

Sommersemester 2016

Softwaretechnik/Software Engineering

http://swt.informatik.uni-freiburg.de/teaching/SS2016/swtvl

Exercise Sheet 6

Early submission: Wednesday, 2016-07-20, 12:00 Regular submission: Thursday, 2016-07-21, 12:00

Exercise 1 – Testing & Coverage Measures (5/10 + 5 Bonus)

Consider the Java function `convert` shown in Figure 1. It is supposed to convert the string representation (decimal, base 10) of an integer to the represented integer value. For example, the string "123" is converted to the integer `123` and the string "-378" is converted to the integer `-378`. 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 $c_0, \ldots, c_{n-1}$, $0 \leq n$ be the 1st, ..., n-th character in `str`. 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 '-'. (Unlike other implementations of string-to-integer conversions, this one should return 0 for the empty string and the string "-".)

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

Recall that submissions need a presentation that is comprehensible enough to be evaluated. Make sure that it is easy to see what the results are, which statements and branches are covered by each test case, etc. (3)

(ii) Modify your test suite from (i) such that it is still **unsuccesful** and still achieves 100% *statement coverage* but achieves **strictly less** than 100% *branch coverage*. (1)

(iii) Recall the shipping calculator binaries from Exercise 5. The inputs were specified as `w` (width), `h` (height), `l` (length), integers ranging from 1 to 250; `Wt` (weight), integer ranging from 1 to 32000 and `type`, a string with poosible values '`metro`', '`interm`' and '`rural`'.

What is the number of test cases required to *exhaustively* test the shipping calculator *inside its specification*?

Assume that you can execute around 1000 tests per second. How many seconds (hours? days? ...) would it then at least take to exhaustively test the shipping calculator using one (single core) CPU? (1)

```
1  int convert(char[] str) throws Exception {
2          if (str.length > 6)
3                  throw new Exception("Length exceeded");
4          if (str.length == 0)
5                  return 0;
6          int number = 0;
7          int digit;
8          int i = 0;
9          if (str[0] == '-')
10                 i = 1;
11         while (i < str.length){
12                 digit = str[i] - '0';
13                 if (digit <= 0 || digit > 9)
14                         throw new Exception("Invalid character");
15                 number = number * 10 + digit;
16                 i = i + 1;
17         }
18         if (str[0] == '-')
19                 number = -number;
20         if (number > 32767 || number < -32768)
21                 throw new Exception("Range exceeded");
22         return number;
23 }
```

Figure 1: Function `convert`

(iv) Consider the expression $expr \equiv x \geq 0 \wedge (y = \texttt{'a'} \vee (z < 0 \wedge v)) \vee \neg(w \neq 100)$ for inputs x:`int`, y:`char`, z:`int`, v:`bool`, w:`int`. Give a test suite that achieves the maximum possible *term coverage* for *expr*. What is the maximum possible term coverage? Justify your answer. (5 Bonus)

Exercise 2 – Verification with PD Calculus (5/10 + 5 Bonus)

Consider the program `multiply` shown in Figure 2. It is annotated with pre- and postconditions and implements integer multiplication as successive addition. The operands are x and y and the result is stored in the variable *result*.

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

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

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

Figure 2: Program `multiply`.

(i) **Give an invariant** for the while loop that enables you to prove the correctness of the program. Apply the rules of the *proof system PD* to **derive a proof** that your invariant is a loop invariant. (2)

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

Note: Please specify which rules or axioms you use at every proof step. Can you reuse the result of task (i)? (3)

(iii) Rewrite the program `multiply` as a C function with the following signature:

```
int multiply( int x, int y );
```

For a start, assume that `multiply` is used in a bigger program where it is only called with values for x between 0 and 15 and values for y between 0 and x, i.e. in the considered bigger program, all callers actually guarantee the precondition

$$\{0 \leq x \leq 15 \wedge 0 \leq y \leq x\}.$$

Annotate your program with the precondition, the postcondition, and the loop invariant (and whatever else you consider necessary) using the VCC syntax for annotations and use VCC[1] to **verify** your annotated program. (2 Bonus)

Hint: Declaring `result` and `i` as local variables of the function makes the task a bit easier. Otherwise, with, e.g., `result` being a global variable, you would need to add a clause `_(writes &result)` to your function declaration.

To enable the tutors to reproduce your results, your annotated C code (with appropriate comments or explanations, if necessary) needs to be part of your submission.

(iv) Discuss the relation of the verification results you obtained from VCC and your result from task (ii). Can you expect VCC to confirm your results? (1 Bonus)

(v) If we verify the considered C program for pre-conditions

$$\{0 \leq x \leq N \wedge 0 \leq y \leq x\}$$

with $N = 15, 150, 1500, 15000$, we observe that the verification time reported by VCC is approximately the same.

Given that successful verification for N corresponds to more than $\frac{1}{2}N^2$ test cases (which is a lot for $N = 15000$), is that measurement plausible? Explain. (1 Bonus)

(vi) Assume you are unsure about your proof from task (ii) and you would like to confirm the result using VCC.

Modify the C program such that it in particular considers the original precondition (cf. Figure 2) 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 with VCC were. Give an interpretation, in your own words, of the output of VCC. (1 Bonus)

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

Exercise 3 – Verification with VCC (10 Bonus)

Recall the shipping calculator program from Tutorial 5. After finding errors through testing, the developers have fixed the program. Now the test suite is unsuccesful. Having seen that there is a very large number of possible inputs for the program, now we would like to use verification –instead of testing– to make sure that the program works correctly.
In the file `calculate_price.c`, you will find the implementation of the function `calculate_price`. It takes as input the following parameters:

- `actual_weight`: an integer between 1 and 32 representing the actual weight of the package in kilograms, after conversion from grams.
- `v_s`: an integer between 1 and $250 \times 250 \times 250/2500$ representing the volumetric weight of the package when using the factor for small packages.
- `v_p`: an integer between 1 and $250 \times 250 \times 250/5000$ representing the volumetric weight of the package when using the factor for parcels.
- `address_type`: a value of type `addrtype` representing the type of the destination address of the package.

The function calculates and returns the price of shipping the package specified in cents (expressed as an integer variable).

(i) Annotate the function `calculate_price` with **preconditions** for the allowable ranges of the parameters, and with **postconditions** that ensure that the price calculation for rules R9 and R10 of the decision table from Tutorial 4 (parcels for rural addresses) are correctly implemented. Verify the program with VCC, document and discuss your results. (5 Bonus)

Hint: you will find that we have already declared the function `max`*. You may also use that function in your specification. Make sure to submit your source code together with your solutions.*

(ii) Add aditional **postconditions** to `calculate_price` to verify that all other rules of the decision table (R2-R8) are also correctly implemented. Verify the program with VCC, document and discuss your results. (5 Bonus)