Prof. Dr. A. Podelski, Dr. B. Westphal
S. Feo Arenis Sommersemester 2015

**Softwaretechnik/Software Engineering**

`http://swt.informatik.uni-freiburg.de/teaching/SS2015/swtvl`

# Exercise Sheet 6

Early submission: Friday, 2015-07-17, 14:00 Regular submission: Monday, 2015-07-20, 14:00

*Hint: Some tasks do not explicitly ask for a proof or a convincing argumentation, not even in the hints. For each of those tasks, consider the possibility that this is* implicitly *required.* ;-) *If you think that this is implicitly required, provide the convincing argumentation, if you think that it is not required, explain why not.*

## Exercise 1 – (CFA Computation Paths) (5/20 Points)

Consider the network $\mathcal{C}$ of CFA given by Figure 1.

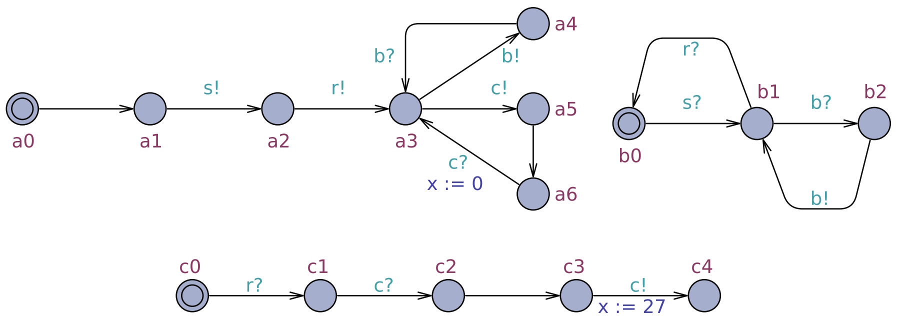

Figure 1: A network of CFA.

(i) Provide the transition graph (cf. Lecture 13, Slide 30) for all reachable configurations according to the (standard) CFA semantics. (4)

*Note: A transition graph consists of complete configurations as nodes (or your abbreviations, appropriately explained/defined); edges represent transitions and are labelled accordingly.*

(ii) Does $\mathcal{C}$ have a deadlock? And a local deadlock? (1)

*Hint: If your answer is 'yes', prove it by giving a computation path and pointing out the (local) deadlock, if your answer is 'no', argue/explain/prove your answer.*

## Exercise 2 – (Verification with Hoare Calculus) (5/20 Points)

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*.
Prove that the program `multiply` computes the multiplication of the operands as specified.

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

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

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

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

Figure 2: Program `multiply`.

(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.* (2)

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

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

**Annotate** your program with the precondition, the postcondition, and the loop invariant (and whatever else you consider necessary) using the VCC syntax for annotations.

*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.*

Use VCC to **verify** your annotated program. Describe what you expect to be the outcome and what the results of the verification with VCC were. Give an interpretation, in your own words, of the output of VCC. (1)

*Hint: 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) Now 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

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

Is the program `multiply` (cf. Figure 2) still partially correct wrt. to the new pre-condition?

*Hint: If yes, provide a proof, if no, provide a counter-example; in both cases with appropriate explanation, of course.*

Modify the C program such that it considers the new precondition and verify it using VCC. **Discuss** whether and why VCC is able to prove correctness with the modified precondition. (1)

**Usage Tips**:
VCC can be used online at the website rise4fun:

```
http://rise4fun.com/vcc
```

You can type your program and run the verification directly on the webpage without having to install the software on your computer.

```java
public int convert(char[] str) throws Exception {
  if (str.length > 6)
    throw new Exception("Length␣exceeded");
  int number = 0;
  int digit;
  int i = 0;
  if (str[0] == '-')
    i = 1;
  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 (str[0] == '-')
    number = -number;
  if (number > 32767 || number < -32768)
    throw new Exception("Range␣exceeded");
  return number;
}
```

Figure 3: Function `convert`

## Exercise 3 – (Testing & Coverage Measures) (5/20 Points)

Consider the Java function `convert` shown in Figure 3. It is supposed to convert the string representation (decimal, base 10) of an integer to the represented integer value.
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 $c_0, \dots, c_{n-1}$, $0 \leq n$ 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 "-".)

**Definition**: Let $\mathcal{T} = \{T_1, T_2, \dots, T_n\}$ be a test suite which achieves $p\,\%$ coverage (in some coverage measure $M$). We call $\mathcal{T}$ *minimal* with respect to $p$ and $M$ if and only if there is no test case $T_i \in \mathcal{T}$ such that $\mathcal{T} \setminus \{T_i\}$ also achieves $p\,\%$ $M$-coverage.

In other words: There are no test cases in the test suite that are *redundant* for achieving $p\,\%$ $M$-coverage.

(i) Give test suites for `convert` with the following properties:

   a) A minimal test suite that achieves 100% *statement coverage*.

   b) A minimal test suite that achieves 100% *branch coverage*.

   c) A test suite that achieves 100% *statement coverage* and 100% *branch coverage* and is minimal for both statement and branch coverage.

   d) A minimal test suite that achieves 100% *statement coverage* but **strictly less than** 100% *branch coverage*.

   The expected value ('soll'-value) should be chosen as precise as possible according to the required functionality of `convert` as described above, i.e. the expected value is not just "does not crash". (2)

   *Hint: The test suites for (a) to (d) need not necessarily be pairwise different.*

(ii) Give a minimal test suite that achieves the maximum possible *term coverage*. What is the maximum possible term coverage and why does your test suite achieve it? (1)

(iii) Provide a modification of `convert` such that the resulting function violates the specification but the error goes undetected when using one of the test suites you provided that achieve 100% *statement coverage*. (1)

   *Hint: Convince your readers of the claim that the resulting function does not satisfy the requirements any more.*

(iv) What is the number of test cases required to *exhaustively* test the function `convert` as given in Figure 3? State your assumptions regarding the platform where the function is to be tested (e.g. the length in bits of the type `char`).

   Assume that executing one test case takes at least one CPU cycle. How many seconds (hours? days? ...) would it then at least take to exhaustively test `convert` using one (single core) 3.9 GHz CPU.

   (1)

*Hint: For all tasks that require providing a* minimal *test suite, provide a proof that the test suite is indeed minimal according to the definition above.*

## Exercise 4 – (Model-based Testing) (5/20 Points)

Assume you are engaged as testing engineer and are requested to test whether an implementation implements a design model. The design model is specified as a communicating finite automaton (CFA).

(i) Provide a test suite that provides 100% edge coverage for the model you chose (see below). An edge is *covered* by a test case if there is a transition in the expected test result that traverses that edge.

   Justify your claim that your test suite provides full coverage by providing a table of test cases vs. covered edges. Point out how you determine full coverage from the information on the table. (3)

(ii) Execute your tests and document the outputs. For each test, state whether it is passed or failed.

   What can you conclude from the execution of your test suite to the question whether the implementation correctly implements the model? (2)

For this task, you may choose to perform testing on the provided implementation of the model of a coin validator for the vending machine (cf. Figure 4), or on (an appropriately improved version of) your implementation of the power window controller model from Exercise Sheet 5. Should you choose to test your own implementation, *instrument* your program to print out the transitions taken (similar to the coin validator implementation), in order to be able to decide whether a test is passed or failed.

A test case can be denoted by the sequence of events input to the implementation and the sequence of transitions expected. E.g., the test case

$$\langle C50, C50; (\mathit{IDLE}, C50, \mathit{HAVE\_C50}), (\mathit{HAVE\_C50}, C50, \mathit{HAVE\_C100})\rangle$$

specifies as input the event sequence $C50, C50$, and the transitions $(\mathit{IDLE}, C50, \mathit{HAVE\_C50})$ and $(\mathit{HAVE\_C50}, C50, \mathit{HAVE\_C100})$ as expected result. Environment conditions: Each test case is supposed to be executed on a freshly started system.
*Hint: If you decide to test your own implementation, please also submit the model of your design and the instrumented source code of your implementation.*

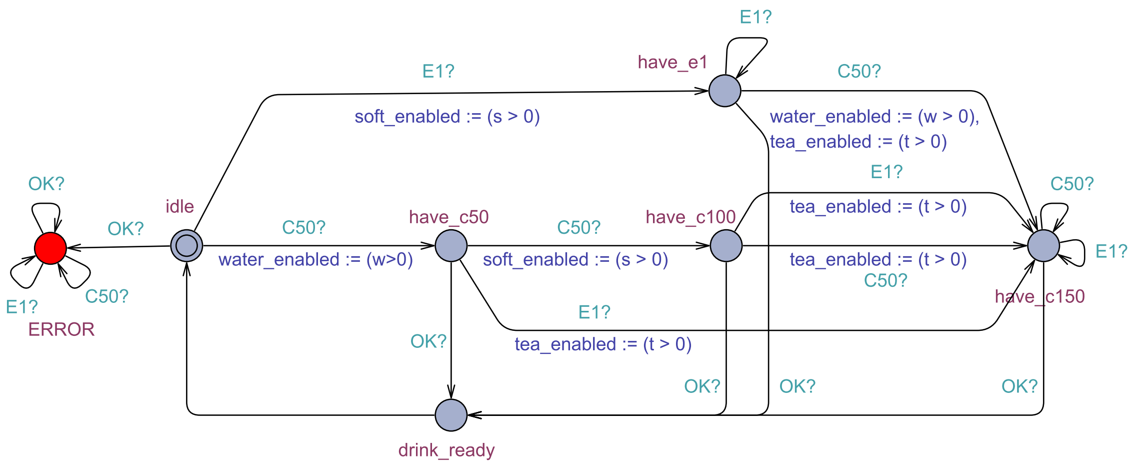

Figure 4: **Model of the coin validator.** The implementation is provided in the file `coinvalidator.jar`. Our implementation of the coin validator allows the test engineer to simulate inputs and reports the transitions taken by the automaton.
Choosing a synchronization event causes the automaton to take the event and perform as many steps as possible before requiring a new input. For example, choosing to send event `OK` in location `HAVE_C50` triggers two transitions: one for the synchronization edge between `HAVE_C50` and `DRINK_READY` and one for the edge between `DRINK_READY` and `IDLE`.

```
----
Current State:  HAVE_C50
----
0:  OK
1:  C50
2:  E1
Input Event:  0
Transitions taken:
(HAVE_C50,OK,DRINK,READY)
(DRINK_READY,TAU,IDLE)
```

**Tips**

You can automate the testing process (on Linux machines) by redirecting the standard input and output streams of the program under test.
The input sequence of a test case can be stored in a text file, with a number per line representing the choice you would make at each step of the program execution.

File `input1.txt`:
```
1
1
```

Don't forget to leave an empty line at the end of the file to ensure that the carriage return (enter) character at the end of the last input is stored in the file.
The file can be used as input to the program under test using the `cat` command as follows:

```
$ cat input1.txt | java -jar coinvalidator.jar
```

The program will run as if each line in the file `input1.txt` were input manually in the console.
If you want to see only the transitions taken by the program, you can use the command:

```
$ cat input1.txt | java -jar coinvalidator.jar | grep "("
```

to filter out the lines that do not contain a transition output.

## Exercise 5 – (Program Verification with VCC) (10 Bonus Points)

You are the programmer in the group in charge of the graphical user interface of a video game. Your current assignment is to implement the calculation of the width in pixels of a *progress bar* (see Figure 5) inside a given window, given a progress percentage value. Knowing the width of the progress bar within the given window will enable other routines to do the rendering.

For your team leader, fidelity of progress bars is extremely important (for reasons not yet known in the team), so you are required to *formally verify* the correctness of your implementation using VCC.

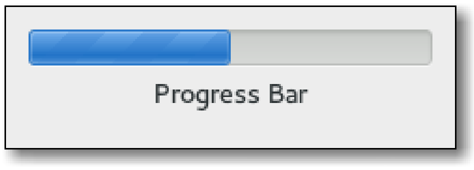

Figure 5: Illustration of a progress bar

You are required to perform the following tasks:

(i) Formalize the assumptions of the function stated below and add them as preconditions of `progress_bar_width` (cf. Figure 6). Shortly explain for each precondition you add, which part of the specification it covers. (5)

(ii) Give an implementation of `progress_bar_width` that calculates the width in pixels of the progress bar and (successfully) verifies with VCC wrt. the existing postconditions and your preconditions. Note that VCC does not support floating point arithmetics, so you may not use `float` or `double` values in your program. (5)

*Hint: formal verification with VCC is usually an iterative process. The first implementation may not verify with the first version of the preconditions. One needs to investigate, whether the preconditions are too weak or the implementation is incorrect and fix things accordingly.*

*Hint: For the development of the implementation, you may use any tools, techniques, skills etc. you like, you are not limited to use VCC as the only tool. You may, e.g., at first create an implementation in your favourite programming language (possibly using assertions to mimic pre- and postconditions) and later port it to C. Recall that the expression language of C is not so different from Java, C++, etc.*

(iii) In order to successfully verify `progress_bar_vector` with VCC, you may need to add assumptions in addition to (1)–(4) (see below) in the form of preconditions. Explain each additional precondition and convince the reader that they are acceptable to be assumed in the usage context of this function. (-1 for each unjustified precondition)

*Hint: the purpose of this "task" is to ensure that the final preconditions are as weak as possible, i.e. that `progress_bar_width` can be used with as many (reasonable) parameter values as possible. The strongest precondition is "false": Then, verification is trivial, but the function is practically unusable because no calling context can ever satisfy this precondition.*

The calculation is performed by the function `progress_bar_width`. The following is assumed:

(1) The allowable window in which to fit in the progress bar is given by two parameters: the left border (`window_left`) and the right border (`window_right`).

(2) The width of this window is larger than zero.

(3) The progress percentage is an integer between 0 and 100

(4) The width of the progress bar is at most the largest width of a commercially available computer monitor. (As an approximation for "the largest", you can use twice (to give room for some future development) the pixels of six 4K displays in a row).

The resulting progress bar width (in pixels) is the return value of the function. A very precise specification of the result value is available:

- When the progress percentage is 100, the progress bar should touch the right border of the window.
- When the progress percentage is less than 100, the progress bar should not touch the right border of the window.
- When the progress percentage is 0, no progress bar should be drawn inside the allowable window.
- When the progress percentage is not 0, at least one pixel should be drawn.
- When the progress percentage is not 0 or 100, the right coordinate of the progress bar should be directly proportional to the progress percentage up to at most one pixel (rounding) tolerance.

  More precisely, it should be in the interval

  $$\left[ \left\lfloor \frac{p \cdot w}{100} \right\rfloor , \left\lceil \frac{p \cdot w}{100} \right\rceil \right]$$

  where $p$ is the progress value and $w$ is the width in pixels of the window.

  *Hint: Given a non-negative real number $x \in \mathbb{R}_0^+$, $\lfloor x \rfloor$ ("floor") ($\lceil x \rceil$ "ceiling") denotes the biggest (smallest) integer smaller (bigger) or equal to $x$, i.e. the biggest (smallest) $n \in \mathbb{N}_0$ such that $n \leq x$ ($x \leq n$). Floor and ceiling of $x \in \mathbb{R}_0^+ \cap \mathbb{N}_0$ are equal to $x$.*

- The width calculated by the function must fit inside the specified allowable window.

The requirements above are already formalized. A function signature, annotated with *postconditions* is shown in Figure 6.

```
int progress_bar_width(int progress, int window_left, int window_right)
_(ensures progress == 0 ==> \result == 0)
_(ensures progress == 100 ==> \result == (window_right - window_left + 1))
_(ensures progress > 0 ==> \result > 0)
_(ensures progress < 100 ==> \result < (window_right - window_left + 1))
_(ensures \result >= ((window_right - window_left + 1) * progress) / 100)
_(ensures \result <= ((window_right - window_left + 1) * progress + 99) / 100)
_(ensures \result >= 0 && \result <= (window_right - window_left + 1))
```

Figure 6: Function signature and postconditions for `progress_bar_width`. The function receives following parameters: `progress` (progress percentage value), `window_left` (the x-coordinate of the left border of the window (in pixels)). `window_right` (the x-coordinate of the right border of the window (in pixels)).