

Teaching Design by Contract using Snap!

Exercise Sheet

1st Marieke Huisman
Formal Methods and Tools
University of Twente
Enschede, The Netherlands
m.huisman@utwente.nl

2nd Raúl E. Monti
Formal Methods and Tools
University of Twente
Enschede, The Netherlands
r.e.monti@utwente.nl

I. ASSERTIONS

For this exercise, load the file *triangles.xml* into Snap!

The following code is supposed to classify triangles with respect to the length of their sides. Given the inputs a , b and c representing the length of each side of a triangle, the code should return 1 if they are all different (a scalene triangle), 2 if only two of the sides are equal (an isosceles triangle), and 3 if they are all equal (an equilateral triangle).



Your tasks for this exercise are the following.

- 1) Call the **triangles** block using the invalid input $a = 0, b = 2, c = 2$. Describe what happens. Why do you think this should not be a valid input?
- 2) Test **triangles** block with different input values of your own. Choose the inputs cleverly so some of them make the assertions fail and some of them don't. For each input that doesn't fail the asserts, write down the output and decide if it is correct.
- 3) Add an assertion to the code to ensure that the final value of r is either 1, 2 or 3.

- 4) The second assertion in the original code partially rules out inputs that do not conform a triangle. The rule is that the length of a single side cannot be bigger than the sum of the two remaining sides. Add to the code the two missing assertions so you rule out all invalid inputs.
- 5) Place the assertion block `assert a!=b or a!=c or b!=c` in at least 2 places where you think it will help you to check if your code works fine. Then test the code with different inputs and check if your assertion is validated or not.

II. POSTCONDITIONS

For this exercise, load the file *max.xml* into Snap!

The following Snap! code returns the maximum of two integers.



In this exercise we will learn about post-conditions. Post-conditions are boolean assertions that describe what is the result of running our code. We can also use them to check if our code works fine. These are your tasks for this exercise:

- 1) It is a fact that the maximum of two numbers is greater or equal to both numbers being compared and at the same time it is equal to at least one of them. Add this facts as predicates in the postcondition (**ensures** slots) of the **max** block. In your post-conditions, make sure you use

the **\result** block whenever you want to talk about the reported value.

- 2) Make a list of tests for your code, this is, a list of pairs of numbers which you can input into the **max** function, that will help you to check if the code works well. Run your code with these inputs. Do all your tests fulfil the post-condition?

III. PRECONDITIONS

For this exercise, load the file *rgb.xml* into Snap!.

Let us now learn about pre-conditions. Pre-conditions are also boolean assertions. They specify conditions on the input we expect for our code. Whoever wants to use your code should take into account this pre-conditions when passing the inputs, and thus we can assume this conditions while developing our code.

In this exercise, you are asked to program a function that given three integers between 0 and 255 representing the red, green and blue colours of a pixel, it returns the value of the brightest, i.e. the one of higher value. As a highly trained developer, you don't want to take this easy. Thus you should take the following steps:

- 1) Think about the conditions on the inputs you will receive, i.e. what do you think would be a valid input. Write down this conditions so you can use them later.
- 2) Do the same with your output. What would be a valid output from your block? Can you for instance relate the value of your output to those of your inputs? Write down a post-condition for your code using this facts.
- 3) Code your **rgb** function in Snap!. Remember to add the pre- and post-conditions. For the pre-conditions use the 'requires' slot beneath your block's name.
- 4) You can now assume your pre-conditions when developing your code and you can check your postconditions with runtime (running Snap!) or static (running Boogie) verification. Try your code in Snap! using different inputs. Then compile your code to Boogie and verify it at <https://rise4fun.com/Boogie/DutchFlag>.

IV. CONTRACTS

For this exercise, load the file *product.xml* into Snap!.

Defining pre- and post-conditions for your code is very important. To stress this, consider the Snap! code of Figure 1 that is supposed to calculate the product of two positive integers:

To grasp on the importance of pre- and post-conditions lets try to test this code with the following inputs:

product	0	10
product	17	91
product	1	0
product	2	-8

As you can see, it is not straight forward to find any mistakes from the first two tests. Although the second test returns an incorrect value, maybe the developer of the code overlooked this mistake and there is no post-condition alerting him about

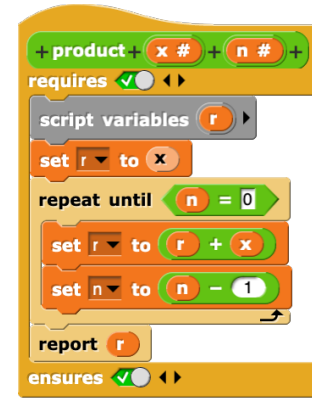


Figure 1. BYOB that multiplies two positive numbers.

it. From the third test case we can see a clear mistake and thus we know that something is wrong. The last case is special, since we use a negative input when our specification only talks about positive inputs. As you can see, the code gets stuck in an infinite loop, and this could have been avoided by correctly defining its pre-conditions.

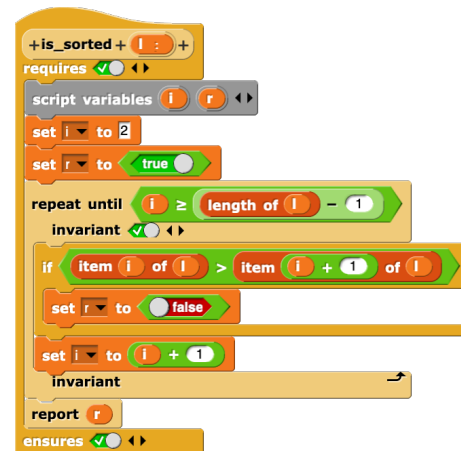
To improve this code do the following:

- Code proper pre-conditions to the code.
- Include the post-condition **r = product \old n x** to hint on the correctness of the result.
- Run the tests again, also feel free to define your own input cases.
- Try to find the mistake in the code, and correct it. Recheck the tests that did not pass.

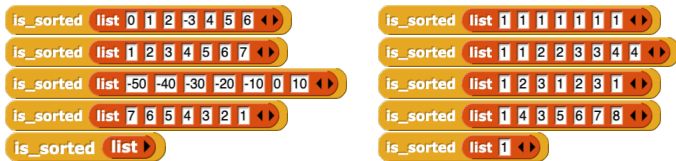
V. STATIC VERIFICATION

For this exercise, load the file *is_sorted.xml* into Snap!.

The following Snap! code is intended to reply if a list is sorted or not:



A common way of analysing the correctness of codes is by running a big list of tests. This tests should not only be a lot, but they should also be intelligently defined as to cover the most variety of inputs possibilities. Let us start this exercise by running a large list of tests on this function:




- 1) Does the code give the correct answer for all these tests?
- 2) Add the following post-condition to the code:



then add the following predicates into the *invariant* slots of your loop (don't worry we will learn about invariants in the next exercise):



- 3) Compile your code to Boogie using the  button on the top right corner of the BYOB coding window. Can you spot your post-conditions in the compiled code? Hint: look for the keyword 'ensures'.
- 4) Copy paste your compiled code in the coding area of the web-page <https://rise4fun.com/Boogie/> and run the Boogie verification by pressing the play button at the bottom. Ask your teacher to help you interpret the result of the verification.
- 5) Try to build a test for which the code fails.



Code bugs are very sneaky. Tests can discover many of them but some others are overseen. Static verification is exhaustive and if the verification passes it means that it passes for any input possible (any input that conforms with the function's requirements). Nevertheless, formal verification is currently difficult to apply to big software projects.

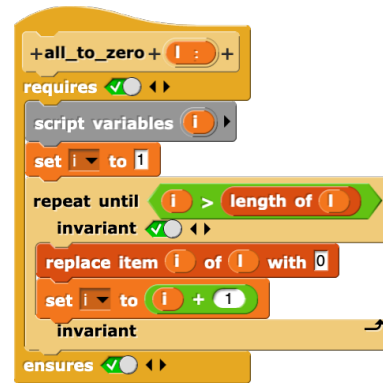
VI. LOOP INVARIANTS

For this exercise, load the file *all_to_zero.xml* into Snap!.

Loops are complex commands. It is not always easy to understand how many times a loop executes and exactly what does it do at each iteration in particular. Invariants in loops allow us to think about their behaviour. An invariant is an assertion, just as a pre- or post-condition. Loop invariants are checked at two places: before the first iteration of the loop and after every iteration of the loop.


Take for instance the following code which loops through a list and sets all its elements to zero:

- 1) As a first exercise define a post-condition for this code in order to check if all the elements in the list have effectively been set to zero. For this, you can make use of the quantification block .
- 2) Also add a post-condition to check that the size of the list has not been changed by the code. Notice that if an input has been modified by the body of your function you can still access it's original value in the post-condition by using the .



- 3) Now design some test cases for your code and run them in Snap!.
- 4) Finally, compile your code to Boogie and verify it. Does it pass the verification? Don't worry if it doesn't.

Most probably your code did not pass the Boogie verification. This may not be because your code is wrong but rather because static verification needs loop invariants to successfully analyse codes with loops. The loop invariant should express a fact that we know that is true before we start the loop and that it remains true just after every loop iteration. For the case of our list this may be something like "all the elements up to, but not including, the current index have been set to zero". Notice that when we finish the loop the current index has the value of the length of the list plus one, thus our invariant will mean that all the elements of the loop have been set to zero, which is what we want to prove in our post-condition. In this way we help the verifier to deduce that the post-condition is valid.

- 1) Using the  block, define a loop invariant as the one we described for our single loop in the code, and place it in the slot next to 'invariant' at the loop header.
- 2) Compile your code to Boogie and run the verification again. Hopefully this time it will pass :)

VII. CRISIS.XML

CRISIS is a multiplayer strategy game where each player tries to conquer the world. For this, each player is initially assigned a set of countries with a number of armies in them. Each player waits for it's turn to choose a limiting country to attack with its armies. The result of the battle is determined by throwing dice, one for each army you own and up to a maximum of four per player. The dice thrown by the attacker are ordered and so are the ones from the defender. Then the highest valued die from the attacker is compared with the highest value die of the defender and an army is removed from the one with lower value. The defender wins the ties. Then the next higher values are compared and so on. If you destroy all armies in an enemy's country you win the country and you move an army into it. The last man standing wins the game.

Accompanying this exercise sheet you will find a file *crisis.xml*. You can open this as a project in Snap!. You will find that the project includes many BYOB blocks in the *control* blocks menu. In order for you to play this game, you should first work on some of this blocks to fix them.

Your first job consists in formally verifying the following BYOB block:



This function should take an input *n* and return a list of *n* integers between 1 and 6 representing the results of throwing *n* dice. Furthermore we should not be able to throw more than 4 dice, given the rules of the game. Your tasks are the following:

- 1) Define a suitable pre-condition for the function to conform to the above specification.
- 2) Define a post-condition which ensures that both the length of the result is the expected and that the values represent the result of throwing dice.
- 3) Define a loop invariant that will allow you to verify this code with static verification.
- 4) Finally compile the block to Boogie and verify it.

You are nearly done! but before you can play you need to work a little on the “compare_dice” block. This block receives two lists with the results of the dice thrown by the attacker and the defender players. This lists should be ordered and the block compares the dice as explained in the introduction to this exercise. Of course you will make as many compares as the size of the shortest list, since one of the list will then run out of dice. For instance suppose the attacker throws (5, 2, 2, 1) and the defender (4, 4); then the comparisons will be 5 vs 4 and 2 vs 4, and both the attacker and the defender will lose one army each. The block returns a list with two elements: the first element is the amount of armies the attacker loses and the second is the amount of armies the defender loses.

Your tasks are the following:

- 1) Fill in the wholes in the block, this is, the empty slots you find around.
- 2) Define proper pre-conditions to the block.
- 3) Verify the block using Boogie. When you manage to pass your verification, describe what guaranties are you obtaining from it, i.e. explain in words your post-conditions,

your loop invariants and under which assumptions this are fulfilled.

Are you done with repairing the CRISIS game? you can now choose the amount of players and start the game :)