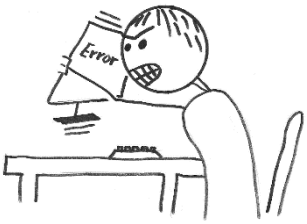
How invariants help writing loops

Author: Sander Kooijmans
Document version: 1.2

Why this document?

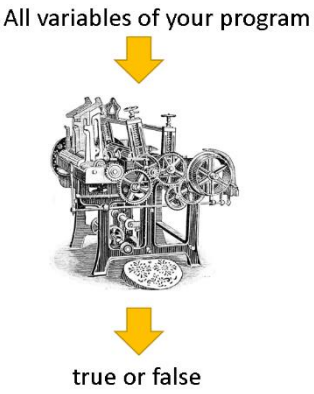


Did you ever feel frustrated because of a nasty bug in your code? Did you spend hours looking at the code and stepping through complex loops with a debugger?

Software engineers must have skills to write correct code. One of these skills is Test-Driven Development (TDD). TDD has received a lot of attention lately, other techniques are overlooked: techniques to prove correctness of algorithms and techniques to derive correct algorithms. In this document I present a technique based on Prof. Dijkstra's style of proving and deriving algorithms.

Predicates

Before explaining invariants I will explain what predicates are. A predicate is a function that has a number of variables as input and returns true or false.



Take the following two statements:

```
int a=4; int b=8;
```

After execution of these two statements the following predicates hold:

```
P(a,b) ≡ a == 4
Q(a,b) ≡ a < b
R(a,b) ≡ 2*a == b
```

For brevity we omit the arguments of a predicate. For example: `R ≡ 2*a == b`

Hoare triple

Tony Hoare invented a technique to prove the correctness of algorithms: Hoare triples. A Hoare triple looks like this: `{Pre} S {Post}`, where `Pre` and `Post` are predicates and S is a statement. The Hoare triple `{Pre} S {Post}` means: when `Pre` holds and S is executed, then if S terminates then `Post` holds. It is important that S terminates. If S does not terminate `Post` does not have to hold.

Hoare used these triples to specify how to prove assignments, if-loops, while-loops and composition.

In this document I will only focus on while-loops.

Hoare triple for a while-loop

What is an invariant? An invariant is a predicate which holds before and after each iteration of a loop. An invariant gives meaning to the variables. And an invariant defines the intent of the variables.

To prove the while-loop `{Inv} while (B) S {Post}` the following four parts must be proven:

1. `Inv` holds before the while-loop is executed
2. `Inv` holds after each iteration
3. `Inv && !B ⇒ Post`
4. The while-loop terminates

`Inv` is called an invariant.

Note that because of part 1 the invariant holds at the beginning of the first iteration. Because of part 2 the invariant holds at the end of the first iteration and therefore at the beginning of the second iteration. And because of part 2 the invariant also holds at the end of the second iteration and therefore at the beginning of the third iteration, and so on. We conclude from parts 1 and 2 that the invariant holds at the beginning and the end of each iteration.

The third part states that the invariant can be used to prove the postcondition. And since the loop has terminated you can use the negation of the guard B as well to prove the postcondition.

The fourth part is obvious. If the loop does never terminate, then there is no guarantee that the postcondition will ever hold.

How can you prove that a while-loop terminates? You have to come up with a bound function. A bound function is a function that has all variables of your program as input and returns a number. The bound function must decrease by at least one each iteration and must be bounded from below. You can compare it with a stairs: if you step down a stairs one or more steps each iteration then sooner or later you will reach the end of the stairs and step on the ground.

Proving an algorithm to calculate 2^n

Let me give an example of a while-loop and to illustrate an invariant and bound function. Here is an algorithm that calculates 2^n for the input variable n. In code I use the notation p^q instead of p^q.

```
// pre: n >= 0
int p=1; int i=0;
// invariant: p == 2^i
// bound function: n-i >= 0
while (i != n) {
   p = 2*p; i = i+1;
}
// post: p == 2^n
```

i	p	2^i
0	1	1
1	2	2
2	4	4
3	8	8
4	16	16
5	32	32

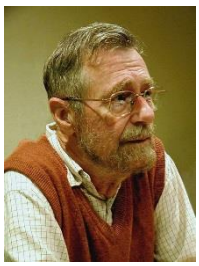
The table to the right shows the values of `i`, p and 2^i before each iteration. Notice that `i` and p change each iteration. Also notice that the p and 2^i are equal in each row in the table.

First check the precondition: it states that n is at least zero. The postcondition states that the variable p is equal to 2^n. The invariant states that p is equal to 2^i. There are four things we have to prove:

1. The invariant holds before the while-loop is executed. Fill in `p = 1` and `i = 0` in the invariant: `1 == 2^0`, which is true.
2. The invariant holds after each iteration. Here we can use the fact that the invariant holds at the beginning of the iteration. So at the beginning holds `p == 2^n`. The assignment `p = 2*p` doubles the value of p. The assignment `i = i+1` doubles the value of `2^i`. The result of the two assignments is that the invariant holds again. Notice that after the first assignment the invariant does not hold. That is not a problem because the invariant holds at the end of the loop.
3. When the loop terminates we know that `i == n`. Together with the invariant `p == 2^i` this yields the postcondition: `p == 2^n`.
4. The bound function `n-i` decreases with exactly one each iteration. The loop ends with `i == n`, thus the bound function cannot get below zero. This proves that the loop terminates.

This was the most formal part of this document. Do not despair if you didn't understand the complete proof. From now on this document will be less formal.

Professor Edsger Wybe Dijkstra



You may know Professor Dijkstra from:

- Eindhoven University of Technology where he has worked in the past (and I graduated from that university)
- Goto statement considered harmful (also called anti-goto letter)
- Semaphores
- Dijkstra's shortest path algorithm
- Winning the Turing Award in 1972

There is one more thing that Professor Dijkstra was famous for: he invented a way to develop proof and program hand in hand.

What I am going to explain you, is inspired by Dijkstra's formal way of developing a proof and program.

Deriving the algorithm to calculate 2^n

Let's write the algorithm to calculate 2^n again, but this time we start with the precondition and postcondition:

```
// pre: n >= 0
…
// post: p == 2^n
```

Next we introduce the invariant. The invariant consists of the postcondition where the constant `n` has been replaced by the new variable `i`.

```
// pre: n >= 0
…
// invariant: p == 2^i
…
// post: p == 2^n
```

The invariant must hold before the loop. So how should we initialize the variables p and `i`? Often the simplest way to initialize is by using 'simple' values like 0 or 1. Can we initialize p with zero? Then we must choose `i` such that 2^i is zero. That is impossible. Thus we must try something else.
The precondition states that the smallest valid value for n is zero. So let's try to initialize `i` with zero. When we choose to initialize `i` with zero then the invariant tells us that p must be initialized with the value 1.

```
// pre: n >= 0
int p=1; int i=0;
// invariant: p == 2^i
…
// post: p == 2^n
```

Next focus on the guard of the while-loop. When should we stop iterating? If `i == n` then the invariant is equal to the postcondition. We must continue iterating until `i == n`. So the guard is `i != n`.

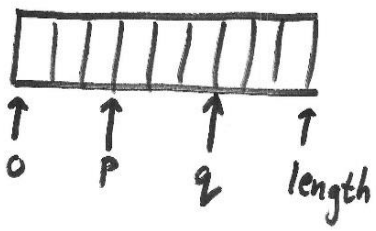
```
// pre: n >= 0
int p=1; int i=0;
// invariant: p == 2^i
while (i != n) {
…
}
// post: p == 2^n
```

Finally we focus on the statements inside the while-loop. We plan on increasing `i`. Increasing `i` by one is the simplest thing we can do. If `i` is increased by 1 then what should we do with p? The right side of the invariant's equation is doubled when `i` is increased by 1. So we need to double the left side as well. This can be done by multiplying p by 2. So the result is:

```
// pre: n >= 0
int p=1; int i=0;
// invariant: p == 2^i
while (i != n) {
   p = 2*p; i = i+1;
}
// post: p == 2^n
```

We already proved that the loop terminates using the bound function `n-i`.

Subarray notation



The second example is about arrays. I use a particular notation for subarrays that works very will with Dijkstra's way of deriving programs. Let us review this notation.

Let a be an array of length 10. Then the following predicates about a hold:

- `a[0..10)` represents the complete array
- `a[0..4)` contains `a[0]`, `a[1]`, `a[2]` and `a[3]`.
- `a[4..6)` contains `a[4]` and `a[5]`.

- Concatenating `a[0..4)` and `a[4..6)` results in `a[0..6)`
- `a[7..8)` contains `a[7]`
- `a[8..8)` is empty
- `a[p..q)` contains `q-p` elements

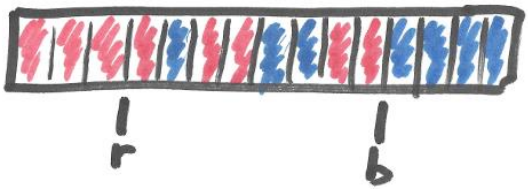
Sorting red and blue elements using swaps

Assume we have an enumeration of colors. It consists of the colors RED and BLUE. Let the variable `colors` be an array containing RED and BLUE elements in an arbitrary order. Let us derive an algorithm that sorts the array `colors` so that it starts with all REDs and ends with all BLUEs. And as extra limitation we only want to use swaps of two elements at a time to modify the array.

We start with writing down the precondition and postcondition:

```
// pre: colors contains red and blue elements in no particular order
…
// post: colors is sorted in the order red, blue
```

The idea I have to sort the array `colors` is as follows: we let the red part 'grow' from left to right and let the blue part 'grow' from right to left. Look at the image at the right. We use two variables to keep track of the bounds of the red and blue parts. `colors[0..r)` contains only red elements, `colors[b..colors.length)` contains only blue elements. The elements in between, thus `colors[r..b)`, still have to be sorted.



Now it is time to formulate the invariant:

```
// pre: colors contains red and blue elements in no particular order
…
// invariant: colors[0..r) are red and colors[b..colors.length) are blue
…
// post: colors is sorted in the order red, blue
```

How do we initialize the while-loop? `r = 1`? Then we must be sure that `colors[0]` is red. And what if `colors.length == 0`? The easiest choice for `r` is zero: `colors[0..0)` contains zero elements. And all of these zero elements are red. And for the same reason we initialize `b` with `colors.length`.

Sounds strange? This is a little bit of mathematics you should know: if you say that 'something' is true for all elements of some set then by definition that 'something' is true for the empty set. Why? Put simply, the empty set does not contain an element that contradicts that 'something'. Compare that to summing all numbers in a set. If the set is empty then the sum is zero. Compare that to multiplying all numbers in a set. If the set is empty then the product is one.

```
// pre: colors contains red and blue elements in no particular order
int r=0; int b=colors.length;
// invariant: colors[0..r) are red and colors[b..colors.length) are blue
…
// post: colors is sorted in the order red, blue
```

When should the while-loop terminate? The `colors[r..b)` still have to be sorted. The while-loop should terminate if that segment is empty. That is the case if `r == b`. Thus the guard of the while-loop is `r != b`.

```
// pre: colors contains red and blue elements in no particular order
int r=0; int b=colors.length;
// invariant: colors[0..r) are red and colors[b..colors.length) are blue
while (r != b) {
…
}
// post: colors is sorted in the order red, blue
```

Now we focus on the body of the while-loop. We want to increase `r` and decrease `b`. When can we increase `r` by one? If `colors[r]` is red. So let's write that down:

```
// pre: colors contains red and blue elements in no particular order
int r=0; int b=colors.length;
// invariant: colors[0..r) are red and colors[b..colors.length) are blue
while (r != b) {
      if (colors[r] == Color.RED) {
            r++;
      }
…
}
// post: colors is sorted in the order red, blue
```

What if `colors[r]` is not red? The precondition tells us that it then must be blue. So we know that `colors[r]` is blue. We don't know anything yet about `colors[b-1]`. What if we swap elements `r` and `b-1` of the array? Then `colors[b-1]` has become blue and we can then decrease `b` by one. And if we decrease `b` by one before executing the swap, then we can swap elements `r` and `b`. That gets rid of the minus one:

```
// pre: colors contains red and blue elements in no particular order
int r=0; int b=colors.length;
// invariant: colors[0..r) are red and colors[b..colors.length) are blue
while (r != b) {
      if (colors[r] == Color.RED) {
            r++;
      } else { // colors[r] == Color.BLUE
            b--;
            swap(colors, r, b);
      }
}
// post: colors is sorted in the order red, blue
```

Does this loop terminate? To answer this question we have to come up with a bound function. Look at the while-loop: each iteration either `r` is increased by one or `b` is decreased by one. This inspires me to come up with the bound function `b - r`. Initially it has the value `colors.length` and it decreases each iteration. The while-loop terminates when `r == b`. Thus `b - r` cannot get below zero. This proves that the while-loop terminates for any input.

When to apply this technique?

Most of the time we write trivial loops. For these loops you don't need to apply this technique. Or maybe you apply this technique in your mind. Within a couple of seconds you convince yourself about the correctness of the loop you just wrote down.

Every now and again a complex loop needs to be written. At those times invariants are a valuable tool to derive and verify your loop.

Exercises

For each exercise derive:

1. Initialization before the loop
2. The guard
3. The assignments in the body
4. Bound function
5. Write tests to check your solution

The exercises start easy and become more and more difficult. Some exercises are marked with the word 'challenge'. Those exercises require more mathematical insight than the other exercises.

Exercise 1: Straightforward loops and empty arrays

Given is an array of integers.

Can you write a loop that calculates the sum of all elements of the array? What is the sum for an empty array?

Can you write a loop that determines the maximum value of the array? What is the maximum value of an empty array?

Can you write a loop that checks whether all numbers in the loop are positive numbers? All numbers of the array `a` are positive if `a[0] > 0 && a[1] > 0 && … && a[a.length-1] > 0`. Are all elements of the empty array positive?

Can you write a loop that checks whether there exists a number in the array that is negative? A negative number exists in the array `a` if `a[0] < 0 || a[1] < 0 || … || a[a.length-1] < 0`. Does a negative number exist in an empty array?

Exercise 2: Another invariant to calculate 2^n

Above a while-loop was derived that calculates 2^n using the invariant `p == 2^i`. Can you derive a while-loop with the slightly different invariant `p*2^i == 2^n`?

How often iterates the while-loop if n is 30? What is the order[1] of your solution?

Exercise 3: A more efficient implementation to calculate powers (challenge)

The goal of this challenge is to write an algorithm that calculates `base^exponent` and has order O(log(exponent)). One way of doing that is by using the following property that holds for `exponent>1`:

$$base^{exponent} = \begin{cases} base(base^2)^{\frac{exponent-1}{2}} & \text{if exponent is odd} \\ (base^2)^{\frac{exponent}{2}} & \text{if exponent is even} \end{cases}$$

Hint: Try as invariant `r * b^e == base^exponent`

Exercise 4: Celebrity problem

In a group of persons a celebrity is someone who is known by everyone but does not know anyone. Can you write an algorithm that finds the celebrity? The number of persons in the group is denoted by `numberPersons`. The persons are represented by numbers in the range `[0..numberPersons)`.

```
// precondition: knows is a two dimensional array such that
// knows[i][j] == "person i knows person j"
…
// postcondition: x is the celebrity.
// That is, for each person j != x holds knows.j.x and !knows.x.j
```

Hint 1: what does `knows.a.b` and `!knows.a.b` tell you about a or b being the celebrity?

Hint 2: try as invariant "`celebrity is in [a..b]`". (If you like try `[a..b)` too and look at the difference in your solutions.)

Exercise 5: Maximum subarray problem

Given an array of integer numbers. Can you can you find the contiguous subarray which has the largest sum? For example, for the sequence of values -2, 1, -3, 4, -1, 2, 1, -5, 4 the contiguous subarray with the largest sum is 4, -1, 2, 1, with sum 6. Find a solution that uses a single while-loop.

[1] The order of an algorithm is denoted by the big O notation. If the algorithm iterates n times, then we say it has order O(n). If an algorithm iterates n^2 we say it has order $O(n^2)$. If the algorithm iterates $3*n^2$ we still say it has order $O(n^2)$. Thus we omit the constant factor 3 from the order. The idea behind this notation is that for large values of n the run time is dominated by the factors of n. For example, in the long run $100*n$ will perform much better than $1*n^2$. Thus an algorithm of order O(n) runs faster than an algorithm of order $O(n^2)$ (for large values of n).

Hint 1: scan through the array values, computing at each position the maximum sum of any subarray ending at that position. This subarray is either empty (in which case its sum is zero) or consists of one more element than the maximum subarray ending at the previous position.

Hint 2: Let's call the array `a`. Let `n` be an integer variable. Use the invariant:
`maxEndingAtN` = "maximum sum of `a[0..n)`, `a[1..n)`, …, `a[n..n)`" and
`maxSoFar` = "maximum sum of any subarray of `a[0..n)`".

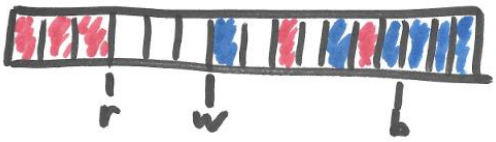
Exercise 6: Dutch national flag problem (sorting three colors)

We continue sorting colors. This time we introduce a third color: white. Why white? Because red, white and blue constitute the Dutch national flag and the author of this document happens to be Dutch...

Let `colors` be an array of red, white and blue elements. Here are the precondition and postcondition:

```
// pre: colors contains red, white & blue elements in no particular order
…
// post: colors is sorted in the order red, white and blue
```

Hint: in the example above we moved the red elements to the start of the array and the blue elements to the end of the array. Can you move the white elements directly after the red elements?



Exercise 7: Quicksort (challenge)

Transform the algorithm from exercise 6 as follows: instead of colors the array should contain numbers. Choose any element in the array. The value of that element is 'white'. All elements with the same value are 'white'. All smaller values are 'red' and all higher values are 'blue'.

See what the transformed algorithm does? It divides the array in three parts. The element that you choose to define the white numbers is called the pivot. The red elements contains all numbers that are smaller than the pivot and all blue elements contain all numbers larger than the pivot.

This looks like a sorting algorithm for numbers. We still have to sort the red and blue segments. For this you need two recursive calls. After adding these two calls you have implemented Quicksort. This algorithm was developed by Tony Hoare (yes, him from the triples) in 1960.

Hint: if the red or blue part contains less than two elements it is already sorted.

If the red and blue part have (about) equal lengths, Quicksort operates in O(n*log(n)). However, if the red or blue part is empty, then it degrades to $O(n^2)$, which is as bad as Bubble Sort and Insertion Sort.

Exercise 8: Binary search

Let `values` be an array containing numbers. This time the array is sorted ascendingly. Can we use the fact that the array is sorted to quickly determine whether some number `n` is present in the array?

Hint 1: which element in the sorted array gives you most information about whether the number `n` is present? The first element, the middle element or the last element?

Hint 2: can you halve the remaining subarray that should contain `n` (if `n` is present) each iteration?

Hint 3: Use as invariant "if `values` contains `n` then it is in `values[low..high)`".

Exercise 9: Primes kata (challenge)

If you are familiar with prime numbers and are in for a challenge you can give this exercise a try. Write a function that calculates the prime factors of a given number. For example: given the number 60 the function should return a list containing 2, 2, 3 and 5 because 2*2*3*5=60. Don't use the sieve of Eratosthenes[2]. Instead use two nested while-loops.

```
List<Integer> factors = new ArrayList<Integer>();
// precondition: p >= 2
…
// postcondition: factors contains the prime factors of p
```

Hint: use the following three invariants:

invariant 1: `factors` contains only primes
invariant 2: n * "product of factors" == p
invariant 3: n has no prime factors less than candidate

[2] The sieve of Eratosthenes finds all prime numbers in the range `[2..n)`. The algorithm starts with the value 2 (which is a prime number) and then marks all multiples of 2 less than n as NOT being a prime number. The multiples of 2 are 4, 6, 8, …. The next number, 3 has not been marked as NOT being prime, thus it is a prime number. The algorithm marks all multiples of 3 less than n as NOT being a prime number. The number 4 is a multiple of 2 and was marked as NOT being a prime number before. The number 5 is not a multiple of 2 and 3 and was not marked as NOT being a prime number. Thus 5 is a prime number. The algorithm continues marking all multiples of 5 as NOT being prime. And so on.

Further reading

Derrick G. Kourie, Bruce W. Watson.
The Correctness-by-Construction Approach to Programming.
Springer, 2012.

Anne Kaldewaij.
Programming: The Derivation of Algorithms
Prentice Hall International (UK) Ltd, 1990.

Tom Verhoeff
http://www.win.tue.nl/~wstomv/edu/nma/
See the section “Summer Session 2006” for material for lectures.

Contact

e-mail: sander.kooijmans@hightechict.nl
website HighTech ICT: www.hightechict.nl
website Sander: www.gogognome.nl