# CITS5501 Software Testing and Quality Assurance
# Semester 1, 2020
# Week 10 workshop – Formal methods

**Exercise 1**

Would you use formal methods for any of the following systems? If so, which systems, and why?

a. The next version of *Confectionery Crush Story*, a web– and mobile-app–based puzzle video game your company develops. The game is available for both Android and iOS mobile devices, and the previous version grossed over $US 1 billion in revenue.

b. *Exemplarex*, software produced for Windows and MacOS operating systems and licensed to educational institutions. The software semi-automatically invigilates exams set by the institutions: machine-learning techniques are used to analyse audio and video of exam candidates to identify possible academic misconduct.

c. The online banking website provided by a major Australian bank, EastPac. Over 5 million customers use the website to perform banking transactions on personal or business bank accounts.

d. A radiation therapy system used to treat cancer patients. The system has two principal components:

   - A radiation therapy machine that delivers controlled doses of radiation to tumour sites, controlled by an embedded software system.
   - A treatment database that includes details of the treatment given to each patient.

**Notes on exercise 1**

The aim of the exercise is just to get you to think about what factors might make formal methods appropriate.

However, some possible answers are:

a. Probably not, because speed-to-market is more important for the games industry
b. Possibly; but if the market doesn't *demand* formal assurance that the software works as specified, it is probably not worth the effort of doing so. In any case, the end result of a machine-learning technique will be some sort of model used for prediction/classification, and formal methods probably can't prove that the classifier is "good".
c. Possibly lightweight formal methods (e.g. advanced type systems) might be appropriate for use in securing web applications. In general, banks apply sufficient security techniques to bring losses through fraud etc to a tolerable level – the cost of fully verifying systems is typically not worth the return.
d. Yes, since this is critical software – lives could be endangered if it malfunctions – the use of formal methods may well be appropriate. Any of the methods we have looked at (advanced type systems, formal verification, model-based methods) could be applied.
   [Note that (in practice*, however, most medical software – even for critical applications – is not, in fact, developed using formal methods.]

**Exercise 2**

Of the formal method techniques we have considered in class, which might be applied to the following systems?

a. Data-analysis software which analyses the results of high-energy physics experiments conducted in particle colliders. Based on data obtained from sensors in the collider, the software attempts to identify *jets* – large numbers of particles all flying in roughly the same direction – using data mining techniques.

   (See here for more details on jets.)

b. A new high-assurance operating system designed to operate voting machines.

c. A database system, which we require should never go into deadlock.

**Sample solutions:**

a. Advanced type system techniques might be used to (e.g.) ensure no mistakes are made in calculations involving physical units.
Model-based techniques (e.g. Alloy-style checking) could be used to "sanity-check" the design of the software before implementation.

b. All techniques might be applied; but formal verification seems especially appropriate. This is because we probably want to ensure that our system behaves exactly as required; we may also want to verify that it satisfies particular security properties. [NB: Note however that, in practice, maintaining security of the physical system and any passwords etc. is paramount]

c. Model-based techniques would be especially appropriate here, since they are well-suited to proving properties such as the absence of deadlock.

### Exercise 3

The following Dafny code is intended to find the position of the largest element of an array. It is only guaranteed to produce a result if the array is *non-empty*, however.

```
method FindMax(arr: array<int>) returns (r: int)
{
  var max_val : int := arr[0];
  var max_idx : int := 0;

  var i : int       := 1;

  while (i < arr.Length)
  {
    if arr[i] > max_val
      {
        max_idx := i;
        max_val := arr[i];
      }
    i := i + 1;
  }
  return max_idx;
}
```

At what points in the code might we insert the following, and what Dafny keywords would be used?

- preconditions
- postconditions
- loop invariants
- assertions

See if you can state what the preconditions and postconditions are (in English is fine).

**Solution:**

Correct locations would be:

```
method FindMax(arr: array<int>) returns (r: int)
   requires /* **preconditions go here** */
   ensures /* **postconditions go here** */
{

  var max_val : int := arr[0];
  var max_idx : int := 0;

  var i : int       := 1;

  /* **assertions could go anywhere in the body** */

  while (i < arr.Length)
    invariant /* **loop invariants go here** */
  {
    if arr[i] > max_val
      {
        max_idx := i;
        max_val := arr[i];
      }
    i := i + 1;
  }
  return max_idx;
}
```

**Challenge exercise**: Try verifying the above code using the online Dafny verifier at https://rise4fun.com/Dafny/tutorial. This will require reading the tutorial, however, in order to learn how to use the `forall` keyword, which we have not covered.

**Sample solution:**

Fully verified code:

```
method FindMax(arr: array<int>) returns (r: int)
   requires arr.Length > 0
   ensures (0 <= r < arr.Length) && (forall k :: 0 <= k < arr.Length
   ↪ ==> arr[r] >= arr[k])
{

  var max_val : int := arr[0];
  var max_idx : int := 0;

  var i : int       := 1;

  assert i >= 1;
  assert max_idx <= i;

  while (i < arr.Length)
    // probably could get away with fewer loop invariants
    invariant 1 <= i <= arr.Length
    invariant 0 <= max_idx  <= arr.Length
    invariant 0 <= max_idx  <= i
    invariant forall k :: 0 <= k <= i-1 ==> max_val >= arr[k]
    invariant (0 <= max_idx < arr.Length) && (max_val == arr[max_idx])
   ↪ && (forall k :: 0 <= k <= i-1 ==> arr[max_idx] >= arr[k])
  {
    if arr[i] > max_val
      {
        max_idx := i;
        max_val := arr[i];
      }
    i := i + 1;
  }
  return max_idx;
}
```