SWEN421 – Lecture 3
Building High Integrity Software with SPARK Ada

Approaches to High Integrity Software

- High level design: module specifications/interfaces/abstract types
- Module design: data structure invariants
- Subprogram design: contracts, loop invariants/variants
- Static correctness checking: static analysis, proof obligations
- Dynamic correctness checking: testing, run-time checks

High Level Design

- System is designed as a collection of components/modules
- Interact with each other via well defined interfaces
- Modules hide information about implementation and data that other modules don’t need access to
- Interface provides data types, public variables and operations on them
- Java interfaces specify names of operations, and types of arguments and results, but no semantics – no difference between a stack and a queue!
- In formal approach, we specify operations formally, in terms of relationships between allowable input and output values
- E.g. if given two ordered sequences, *merge* will produced an ordered sequence containing all of the elements of both inputs
- What about repeated elements???

Programming with Modules

- An Ada package (module) has two parts: specification and body
- Other modules can only see the specification
- Private part of specification declares types, etc. needed in the public part
- Formal specifications allow us to prove correctness of top level design, before components have been implemented
- Dependency contracts provide finer control over what each operation can access and/or update (later)

Programming by successive refinement

- We can develop a system in several steps, from an abstract specification through more and more concrete layers till we reach a final implementation
- Can prove correctness of each refinement/layer relative to the one above it, rather than proving correctness of implementation wrt abstract specification in one step

Specification facilities in Spark Ada

- Ada has two mechanisms for adding meta-level information to programs:
- Pragmas were in original Ada, for providing compiler operations, hardware specific details, etc.
  - pragma name(value);   eg: pragma SPARK_Mode(On);
- Aspects were added in Ada 2012, for providing other kinds of program annotations
  - with name => definition   eg: with SPARK_Mode => On
- Some things can be defined using either.
- Ex: Look up the lists of pragmas and aspects available!

Contracts for subprograms

- Subprograms are specified in terms of:
  - Precondition defining allowable inputs
  - Postcondition defining allowable out put for any allowable input
  - In Ada these are given as aspects, Pre and Post
    (cf requires and ensures in Whiley)
- procedure Sqrt(x: in Integer; z: out Integer)
  with Pre => x >= 0,
  Post => z**2 <= x and (z+1)**2 > x;
- Function Min(x, y: in Integer) return Integer
  with Post => (Abs’Result = x or Abs’Result = y) and
  Abs’Result >= x and Abs’Result >= y

Contracts for subprograms

- Contracts need not specify all relevant details
- ```
  procedure Insert1(x: in Integer; a: in array(<>) Integer; b: out array(<>) Integer)
    with Post  =>  b'Length = a'Length+1;
  ```
- ```
  procedure Insert2(x: in Integer; a: in array(<>) Integer ; b: out array(<>) Integer)
    with Pre  =>  (for all i in a'First .. a'Last-1 => a(i) <= a(i+1)),
         Post  =>  (for all i in b'First .. b'Last-1 => b(i) <= b(i+1));
  ```
- ```
  procedure Insert3(x: in Integer; a: in array(<>) Integer ; b: out array(<>) Integer)
    with Pre  =>  (for all i in a'First .. a'Last-1 => a(i) <= a(i+1)),
         Post  =>  (for all i in b'First .. b'Last-1 => b(i) <= b(i+1)) and
                   (for some k in b'Range =>
                     (for all i in b'First .. k-1 => b(i) = a(i)) and
                     b(k) = x and
                     (for all i in k+1 .. b'Last => b(i) = a(i-1)))
  ```

- Pre and postconditions often need to use quantifiers
  - Use specification functions to capture abstractions and simply specifications: ord(a)
  - Use collections to express specifications more abstractly (later)
  ```