Dafny troubleshooting guide

Tips to write better specifications

- **Know your Boolean operators!**
  - If the method must satisfy different post-conditions based on the input value, use an implication to describe those conditional behaviors
    `ensures P ==> Q1`
    `ensures !P ==> Q2`
    The condition P should be based on values before the execution (and therefore use `old`), otherwise these assertions might not mean what you think they do.
  - Avoid writing `b == true` and `b == false` when testing Boolean values, prefer `b` and `!b` instead.
  - If a condition must be satisfied if and only if another condition is true (for example “the method returns true when the element has been added to the data structure”) use an equivalence <==> rather than multiple implications.
- **Use the right data types**
  - Like Haskell or ML, Dafny has algebraic data types that can be used (among other things) for enumerations or tuples.
  - There are also built-in types for sets, multisets, sequences, maps… Use them to your advantage and don’t limit yourself to arrays.
  - Choosing the right data type not only makes the specification much easier to read, but data types are in themselves a form of specification!
- **Keep it small and simple**
  - In the modifies clause, include only what is actually modified and nothing else. Avoid `modifies this` as it means that any field of this object can be modified. Instead detail exactly the fields that are modified, and only those.
  - Keeping things simple is useful when coding, but even more important for specification. Your specification should be complete, but if it is complex and unreadable it is not likely to be useful (or correct).

Common error messages and how to deal with them

`call may violate context's modifies clause`

This error can be caused by a method writing in a location that is not included in its "modifies" clause. However if you get it in your main program, it is likely because a method initialized a field without ensuring that it was a fresh location. The explanation is that the prover cannot know by default where the field points to, so the next time you modify it, it assumes that this could affect any location in the memory.

The fix: for any method or constructor that initializes a field f with a call to `new`, make sure that the specification includes `ensures fresh(f)`. With this indication the prover knows that the field refers to a memory location that is not shared with any other object.

`assertion violation`

This error could mean that there exists conditions under which an assertion (signaled by the command assert or ensures) is violated. But it could also mean that the prover simply does not have enough information to prove the assertion. When the prover encounters a call to a method (in the main program or inside another method), it is "blind" to the method body, and instead only uses the information given by the contract of the method, which is already proven separately. This allows the prover to be more efficient and ensures a modular design.

The fix: make the post-conditions of your methods stronger, so that they describe exactly what the method does.

`index out of range`

The prover checks that array accesses are only done within the bounds of the array, both in the program body and in the assertions. In order to do that it must have enough information about the indices and how they relate to the array length.

The fix: make the pre-conditions of your methods stronger, in particular concerning the indices. If an index is modified, indicate its new value in the post-condition. In post-conditions, make sure that you do not confuse the value of a index i before the method `old(i)` and its value after, `i`.

`array may be null`

The prover also checks whether arrays have been initialized. This is usually straightforward but if your "modifies" clause is too generous, the prover may assume that a previously initialized array reference has been modified.

The fix: make your modifies clause as precise as possible. Be particularly careful with `modifies this`, which means that any field of the object may be assumed to have been modified.