6.6

* Always respect the implied unit systems, when converting explicitly from one numeric type to another.

6.9

* Do not suppress the checks provided by the language.

6.11

* Do not use the features explicitly identified as unsafe.

6.14

* Write explicit checks for null values to avoid exceptions being raised.
* Handle exceptions raised by attempts to dereference null values.

6.17

* Avoid the use of similar names to denote different objects of the same type.

6.18

* Use Ada compilers that detect and generate compiler warnings for dead stores.
* Use static analysis tools to detect such problems.

6.25

* Consider using short-circuit forms by default (errors resulting from the incorrect use of short-circuit forms are much less common), thought this can make it more difficult to express the distinction between the cases where short-circuited evaluation is known to be needed (either for correctness or for performance) and those where it is not.

6.35

* Alternatively, monitor the depth of the recursion such as by passing a recursion depth value that is incremented for each level of recursion, and use a subtype constraint or explicit comparison against a maximum depth limit to trigger handling of the situation.

6.39

* Use a completely static model where all storage is allocated from global memory and explicitly managed under program control.

<<<6.41.has a duplicated “Follow the mitigation…. >>>

6.44

* Preceed downcasts by a class-wide membership test as needed to avoid possible exceptions.
* Use type invariants where allowed to detect semantic violations caused by upcasts.

6.50

* Put appropriate exception handlers in all routines that call library routines, including the catch-all exception handler **when others** =>.
* Put appropriate exception handlers in all routines that are called by library routines, including the catch-all exception handler **when others** =>.

6.52

* Do not suppress language defined checks.
* If language-defined checks must be suppressed, use static analysis to prove that the code is correct for all combinations of inputs.
* If language-defined checks must be suppressed, use explicit checks at appropriate places in the code to ensure that errors are detected before any processing that relies on the correct values.

6.55

* For situation where order of evaluation or number of evaluations is unspecified, use only operations with no side-effects, or idempotent behaviour, to avoid the vulnerability.

6.57

* Minimize use of any predefined numeric types, as the ranges and precisions of these are all implementation defined. Instead, declare your own numeric types to match your particular application needs.

6.60

* Where possible do not use forced termination

6.62

* If possible, do not use the abort feature

6.63

* On a single processor, make use of a scheduling regime based on ceiling protocols, this is guaranteed to be deadlock free (if the tasks and resources are assigned the correct priorities – a static property that can be checked offline).
* For multicore, consider assigning all interacting tasks to the same CPU then treat each such group as a separate independent entity.
* Minimize the use of dynamic priorities and dynamic ceiling priorities (so that the static values can be verified)