SPARK 2014: Test and Proof

University.adacore.com

Various Combinations of Tests and Proofs

Overall context is functional verification of code

Combination can take various forms:

- **Test before Proof** – contracts used first in test, possibly later in proof
- **Test for Proof** – contracts executed in test to help with development of proof
- **Test alongside Proof** – some modules are tested and other modules are proved
- **Test as Proof** – exhaustive test as good as proof
- **Test on top of Proof** – proof at unit level completed with test at integration level, also using contracts

Test (be)for(e) Proof – Activating Run-time Checks

Need to activate run-time checks in executable

Constraint_Error exceptions activated by default

- Use -gnat-p to revert effect of previous -gnatp (say in project file)
- Use -gnato to activate overflow checking (default since GNAT 7.3)

Special handling of floating-point computations

- Use -gnateF to activate bound checking on standard float types
- Use -msse2 -mfpmath=sse to forbid use of 80bits registers and FMA on x86 processors
- Runtime/BSP should enforce use of Round-Nearest-tie-to-Even (RNE) rounding mode

Test (be)for(e) Proof – Activating Assertions

Need to activate assertions in executable

Assertion_Error exceptions deactivated by default

- Use -gnata to activate globally
- Use pragma Assertion_Policy to activate file-by-file
- Use -gnateE to get more precise error messages (Contract_Cases)

Special assertions checked at run time

- Contract_Cases → checks one and only one case activated
- Loop_Invariant → checks assertion holds (even if not inductive)
- Assume → checks assertion holds (even if not subject to proof)
- Loop_Variant → checks variant decreases wrt previous iteration

Test (be)for(e) Proof – Activating Ghost Code

Need to activate ghost code in executable

Ghost code, like assertions, is deactivated by default

- Use -gnata to activate globally
- Use pragma Assertion_Policy (Ghost => Check) to activate locally

Inconsistent combinations will be rejected by GNAT

- Ignored ghost entity in activated assertion
- Ignored ghost assignment to activated ghost variable

Test for Proof – Overflow Checking Mode

Problem: ignore overflow checks in assertions/contracts

- Only applies to signed integer arithmetic
- Does not apply inside an expression function returning an integer

Solution: use unbounded arithmetic in assertions/contracts

- Will use 64bits signed arithmetic when sufficient
- Otherwise use a run-time library for unbounded arithmetic

Two ways to activate unbounded arithmetic

- Use -gnato13 compiler switch
- Use pragma Overflow_Mode with arguments (General => Strict, Assertions => Eliminated) in configuration pragma file

Test alongside Proof – Checking Proof Assumptions

Need to check dynamically the assumptions done in proof

- Postcondition of tested subprogram called in proved subprogram
- Precondition of proved subprogram called in tested subprogram

Other assumptions beyond pre- and postconditions

- Global variables read and written by tested subprogram
- Non-aliasing of inputs and outputs of proved subprogram
- No run-time errors in tested subprogram

GNATprove can list assumptions used in proof

- Switch --assumptions adds info in gnatprove.out file

See *Explicit Assumptions - A Prenup for Marrying Static and Dynamic Program Verification*

Test alongside Proof – Rules for Defining the Boundary

SPARK_Mode defines a simple boundary test vs. proof

- Subprograms with SPARK_Mode(On) should be proved
- Subprograms with SPARK_Mode(Off) should be tested

SPARK_Mode can be used at different levels

- Project-wise switch in configuration pragma file (with value On) → explicit exemptions of units/subprograms in the code
- Distinct GNAT project with SPARK_Mode(On) for proof on subset of units
- Explicit SPARK_Mode(On) on units that should be proved

Unproved checks inside proved subprograms are justified

- Use of pragma Annotate inside the code

Test alongside Proof – Special Compilation Switches

Validity checking for reads of uninitialized data

- Compilation switch -gnatVa enables validity checking
- pragma Initialize_Scalars uses invalid default values
- Compilation switch -gnateV enables validity checking for composite types (records, arrays) → extra checks to detect violation of SPARK stronger data initialization policy

Non-aliasing checks for parameters

- Compilation switch -gnateA enables non-aliasing checks between parameters
- Does not apply to aliasing between parameters and globals

Test as Proof – Feasibility of Exhaustive Testing

Exhaustive testing covers all possible input values

- Typically possible for numerical computations involving few values
- e.g. OK for 32 bits values, not for 64 bits ones
  - binary op on 16 bits → 1 second with 4GHz
  - unary op on 32 bits → 1 second with 4GHz
  - binary op on 32 bits → 2 years with 64 cores at 4GHz
- In practice, this can be feasible for trigonometric functions on 32 bits floats

Representative/boundary values may be enough

- Partitioning of the input state in equivalent classes
- Relies on continuous/linear behavior inside a partition

Test on top of Proof – Combining Unit Proof and Integration Test

Unit Proof of AoRTE combined with Integration Test

- Combination used by Altran UK on several projects
- Unit Proof assumes subprogram contracts
- Integration Test verifies subprogram contracts

Unit Proof of Contracts combined with Integration Test

- Test exercises the assumptions made in proof
- One way to show Property Preservation between Source Code and Executable Object Code from DO-178C/DO-333
  - Integration Test performed twice: once with contracts to show they are verified in EOC, once without to show final executable behaves the same

Quiz

Is this correct? 1/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

I am stuck with an unproved assertion. My options are:

(a) switch --level to 4 and --timeout to 360

(b) open a ticket on GNAT Tracker

(c) justify the unproved check manually

Is this correct? **1/10**

NO

I am stuck with an unproved assertion. My options are:

(a) switch --level to 4 and --timeout to 360

(b) open a ticket on GNAT Tracker

(c) justify the unproved check manually

Why not, but only after checking this last option:

(d) run tests to see if the assertion actually holds

Is this correct? 2/10

YES
(click on the check icon)

NO
(click on the error location(s))

The same contracts are useful for test and for proof, so it’s useful to develop them for test initially.

Is this correct? 2/10

NO

The same contracts are useful for test and for proof, so it’s useful to develop them for test initially.

In fact, proof requires more contracts that test, as each subprogram is analyzed separately.

But these are a superset of the contracts used for test.

Is this correct? 3/10

YES
(click on the check icon)

NO
(click on the error location(s))

Assertions need to be activated explicitly at compilation for getting the corresponding run-time checks.

Is this correct? 3/10

✓ YES

Assertions need to be activated explicitly at compilation for getting the corresponding run-time checks.

Correct. Use switch -gnata to activate assertions.

Is this correct? 4/10

YES
(click on the check icon)

NO
(click on the error location(s))

When assertions are activated, loop invariants are checked to be inductive on specific executions.

Is this correct?

4/10

✗ NO

When assertions are activated, loop invariants are checked to be inductive on specific executions.

No, loop invariants are checked dynamically exactly like assertions.

The inductive property is not something that can be tested.

Is this correct? 5/10

✓ YES
(click on the check icon)

✗ NO
(click on the error location(s))

Procedure P which is proved calls function T which is tested. To make sure the assumptions used in the proof of P are verified, we should check dynamically the precondition of T.

Is this correct? 5/10

✖ NO

Procedure P which is proved calls function T which is tested. To make sure the assumptions used in the proof of P are verified, we should check dynamically the precondition of T.

No, the precondition is proved at the call site of T in P.

But we should check dynamically the postcondition of T.

Is this correct? 6/10

YES
(click on the check icon)

NO
(click on the error location(s))

Function T which is tested calls procedure P which is proved. To make sure the assumptions used in the proof of P are verified, we should check dynamically the precondition of P.

✓ YES

Is this correct? 6/10

Function T which is tested calls procedure P which is proved. To make sure the assumptions used in the proof of P are verified, we should check dynamically the precondition of P.

Correct. The proof of P depends on its precondition being satisfied at every call.

Is this correct? 7/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

However procedure P (proved) and function T (tested) call each other, we can verify the assumptions of proof by checking dynamically all preconditions and postconditions during tests of T.

Is this correct? 7/10

NO

However procedure P (proved) and function T (tested) call each other, we can verify the assumptions of proof by checking dynamically all preconditions and postconditions during tests of T.

That covers only functional contracts. There are other assumptions made in proof, related to initialization, effects and non-aliasing.

Is this correct? 8/10

YES
(click on the check icon)

NO
(click on the error location(s))

Proof is superior to test in every aspect.

Is this correct? 8/10

NO

Proof is superior to test in every aspect.

Maybe for the aspects Pre and Post.

But not in other aspects of verification: non-functional verification (memory footprint, execution time), match with hardware, integration in environment…

And testing can even be exhaustive sometimes!

Is this correct? 9/10

YES
(click on the check icon)

NO
(click on the error location(s))

When mixing test and proof at different levels, proof should be done at unit level and test at integration level.

Is this correct? 9/10

NO

When mixing test and proof at different levels, proof should be done at unit level and test at integration level.

This is only one possibility that has been used in practice.

The opposite could be envisioned: test low-level functionalities (e.g. crypto in hardware), and prove correct integration of low-level functionalities.

Is this correct? 10/10

YES
(click on the check icon)

NO
(click on the error location(s))

There are many ways to mix test and proof, and yours may not be in these slides.

✓ YES

? Is this correct? 10/10

There are many ways to mix test and proof, and yours may not be in these slides.

YES! (and show me yours)

university.adacore.com