From learning examples to High-Integrity Middleware

Frama-C and SPARK day 2017

Christophe Garion and Jérôme Hugues

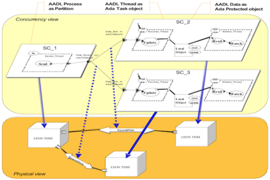
ISAE-SUPAERO – DISC

Outline

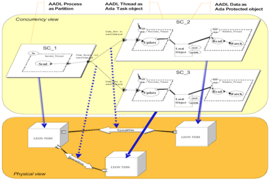
1. **The PolyORB-HI runtime**
2. **SPARK by Example**

AADL, Ocarina and PolyORB-HI

AADL is an architectural design language aimed at embedded systems.
It allows to define software and hardware components.



AADL is an architectural design language aimed at embedded systems.
It allows to define software and hardware components.

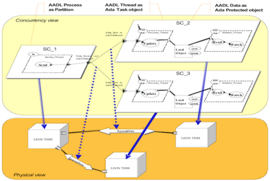


Ocarina is an AADL model processor targetting both C RTOS or Ada via native or Ravenscar targets

C source   Ada source

AADL, Ocarina and PolyORB-HI

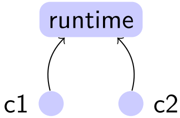
AADL is an architectural design language aimed at embedded systems.
It allows to define software and hardware components.



Ocarina is an AADL model processor targetting both C RTOS or Ada via native or Ravenscar targets

C source   Ada source

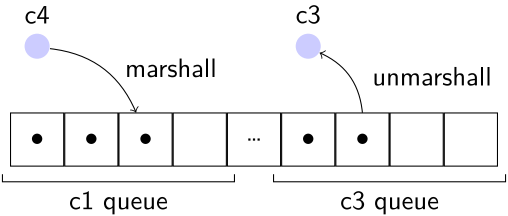
PolyORB-HI is a high-integrity runtime with a C and an Ada implementation.



PolyORB-HI services

Services offered by PolyORB-HI:

- types and time management
- marshalling and unmarshalling facilities
- messages management
- a global queue to exchange messages between components
- patterns for periodic, sporadic tasks etc.



PolyORB-HI proof

Proof of **both runtimes** (C and Ada versions):

- absence of runtime errors
- contract correctness
- using Frama-C for C and SPARK2014 for Ada

PolyORB-HI proof

Proof of **both runtimes** (C and Ada versions):

- absence of runtime errors
- contract correctness
- using Frama-C for C and SPARK2014 for Ada

But:

- the runtimes are already written (by good C and Ada programmers)
- we are adding the contracts without retroengineering the code (at least we are trying...)

PolyORB-HI proof

Proof of **both runtimes** (C and Ada versions):

- absence of runtime errors
- contract correctness
- using Frama-C for C and SPARK2014 for Ada

But:

- the runtimes are already written (by good C and Ada programmers)
- we are adding the contracts without retroengineering the code (at least we are trying...)

Moreover, some parts of the contracts depend on the AADL model: number of tasks, etc.

➥ how to fix these numbers to be representative?

Status of PolyORB-HI/Ada

PolyORB-HI/Ada leverages Ada 2012 High-Integrity profile

- arrays as first class citizen $\rightarrow$ no pointers!
- sizes of all messages known from the model $\rightarrow$ all arrays are statically bounded, no dynamic allocation!
- generics $\rightarrow$ adaptation to user-defined types made easy!
- concurrency built in SPARK 2014, using Ravenscar $\rightarrow$ deterministic and provable tasking!

Annotations generated to ensure compliance with SPARK language, proper initialization of all elements and absence of run-time errors, and annotation of key integrity property of core elements (message queues and buffer management), ensuring a Gold level !

More on `https://github.com/OpenAADL/polyorb-hi-ada` (check the `spark2014` branch).

PS: this slide has been writing by a Ada/SPARK enthusiast ☺

Status of PolyORB-HI/C

More difficult for PolyORB-HI/C:

- good C programmers have implemented the runtime, so they use `void *` pointers, unions etc.
- absence of runtime errors can be easily discharged using correct preconditions (see previous remark)
- functional correctness is more difficult:
  - we have found one (minor) bug!
  - proof implies some major refactoring of code (for instance unions)
  - `void *` pointers are problematic
  - concurrency problems between tasks have not been tackled yet...
  - some automatic proofs are very long

More on `https://github.com/OpenAADL/polyorb-hi-c` (check the various `acsl` branches).

Outline

Why SPARK by Example?

We want our students to work on the PolyORB-HI projects, but time dedicated to research projects is short at ISAE-SUPAERO (roughly 2 months).

Good complete references are available for both languages:

- ACSL Frama-C implementation
- Frama-C user Manual
- WP manual

- SPARK 2014 User's Guide
- Building High Integrity Applications with SPARK

Why SPARK by Example?

We want our students to work on the PolyORB-HI projects, but time dedicated to research projects is short at ISAE-SUPAERO (roughly 2 months).

Good complete references are available for both languages:

- ACSL Frama-C implementation
- Frama-C user Manual
- WP manual
- SPARK 2014 User's Guide
- Building High Integrity Applications with SPARK

We offer to 3rd year students attending the Critical Embedded Systems track a course on formal methods in which they have to develop a small string library.

➥ they use "ACSL by Example" a lot to learn ACSL through classical algorithms!
See `https://gitlab.fokus.fraunhofer.de/verification/open-acslbyexample`

SPARK by Example: the contract

The idea:

- provide a booklet in the spirit of "ACSL by Example" in which students can find classical algorithms and learn SPARK "hands-on"
- start from each function presented in "ACSL by Example"
- write a SPARK version of this function, first by translating the C function signature and then by trying to "SPARKify" the function
- compare both approaches

A good guinea pig: me!

- minimal knowledge of Ada
- rather good knowledge of C and Frama-C
- will do/have done all possible mistakes and clumsiness in SPARK

SPARK by Example: an example

The `Equal` and the `Mismatch` functions are defined as follows:

- `Equal` verifies if two arrays are equal
- `Mismatch` returns the first index at which two arrays differ

Let us look at their specification and implementation in ACSL by example.

First, a predicate is specified to define what means "array `a` and array `b` are equal":

```
predicate
  EqualRanges{K,L}(value_type* a, integer m, integer n,
                   value_type* b, integer p) =
    \let s = n - m;
    \forall integer i; 0 <= i < s ==> \at(a[m+i],K) == \at(b[p+i],L);
```

Several overloaded versions of the predicate are also defined.

C: Mismatch specification

`Mismatch` can be easily specified using `EqualRanges`:

```
requires valid:  \valid_read(a + (0..n-1));
requires valid:  \valid_read(b + (0..n-1));

assigns \nothing;

behavior all_equal:
  assumes EqualRanges{Here,Here}(a, n, b);
  ensures result: \result == n;

behavior some_not_equal:
  assumes !EqualRanges{Here,Here}(a, n, b);
  ensures bound:   0 <= \result < n;
  ensures result:  a[\result] != b[\result];
  ensures first:   EqualRanges{Here,Here}(a, \result, b);

complete behaviors;
disjoint behaviors;
```

C: mismatch implementation

Finally, `Mismatch` is implemented straightforwardly:

```
size_type
mismatch(const value_type* a, size_type n, const value_type* b)
{
  /*@
    loop invariant bound:  0 <= i <= n;
    loop invariant equal:  EqualRanges{Here,Here}(a, i, b);
    loop assigns i;
    loop variant n-i;
  */
  for (size_type i = 0; i < n; i++) {
    if (a[i] != b[i]) {
      return i;
    }
  }

  return n;
}
```

Defining predicates with SPARK

To define predicates with SPARK, ghost functions and expressions can be used:

```
function Equal_Ranges (A : T_Arr; Offset_A : Natural; Size_A : Natural;
                       B : T_Arr; Offset_B : Natural)
                      return Boolean is
  (for all I in 0 .. Size_A - Offset_A - 1 =>
     A(A'First + I) = B(B'First + Offset_B + I));
```

but as such functions are also verified by `gnatprove`, preconditions must be added to prove that no overflow or index check may fail:

```
function Equal_Ranges (A : T_Arr; Offset_A : Natural; Size_A : Natural;
                       B : T_Arr; Offset_B : Natural) return Boolean is
  (for all I in 0 .. Size_A - Offset_A - 1 =>
    A(A'First + Offset_A + I) = B(B'First + Offset_B + I))
  with Pre => Size_A <= A'Length and then
              Offset_A < A'Length and then
              B'Length >= Size_A and then
              Offset_B <= B'Length - Size_A + Offset_A and then
              Offset_B < B'Length;
```

Defining predicates with SPARK

Easier: use equality on arrays with no limited types provided by Ada:

```
function Equal_Ranges (A : T_Arr; B : T_Arr) return Boolean is
  (A = B);
```

Consequence: no predicate is needed, simply use = or a simplified version of `Equal_Ranges` with a slice (with SPARK Pro 17):

```
function Equal_Ranges (A : T_Arr; B : T_Arr; Offset : Natural)
  return Boolean is
  (A(A'First .. A'First + Offset) = B(B'First .. B'First + Offset))
with
  Pre => Offset < A'Length and then
         Offset < B'Length;
```

SPARK: specifying Mismatch

`Mismatch` is specified with contract cases, completeness and disjointness is automatically checked:

```
function Mismatch (A : T_Arr; B : T_Arr) return Natural with
  Pre => A'Length <= B'Length,
  Contract_Cases => (
    A = B (B'First .. B'First - 1 + A'Length) =>
      Mismatch'Result = A'Length,
    others                                    =>
      (A (A'First + Mismatch'Result) /= B (B'First + Mismatch'Result))
      and then
      (if (Mismatch'Result /= 0) then
       Equal_Ranges.Equal_Ranges(A, B, Mismatch'Result - 1)));
```

SPARK: implementing Mismatch

`Mismatch` is classically implemented. Notice that we do not need to specify an invariant for variable bounds or frame condition:

```
function Mismatch (A : T_Arr; B : T_Arr) return Natural is
begin
   for I in 0 .. A'Length - 1 loop
      if (A (A'First + I) /= B (B'First + I)) then
         return I;
      end if;

      pragma Loop_Invariant
        (Equal_Ranges.Equal_Ranges (A, B, I));
      pragma Loop_Variant
        (Increases => I);
   end loop;

   return A'Length;
end Mismatch;
```

VC verifications for Mismatch

Some results for `Mismatch`:

Frama-C Silicon		
postconditions	4	AE
loop invariants	4	AE
loop variant	2	AE + Qed
assigns	5	Qed
behaviors	2	Qed
RTE	2	AE

SPARK Pro 17	
contract cases	2
loop invariant	2
loop variant	1
preconditions	2
RTE	31

- for functional correctness proof, time is quasi equivalent
- overflow, index and ranges checks are numerous in SPARK due to the language
- "extra specifications" in Frama-C are easily discharged by Qed

Writing Equal using Mismatch

In ACSL by Example, Equal is written using Mismatch. We can use Mismatch as an implementation or a specification:

```ada
function Direct_Equal (A : T_Arr; B : T_Arr) return Boolean is
  (A = B (B'First .. B'First - 1 + A'Length))
with
  Pre  => A'Length <= B'Length,
  Post => (Direct_Equal'Result =
             (Mismatch.Mismatch (A, B) = A'Length));

function Equal (A : T_Arr; B : T_Arr) return Boolean is
  (Mismatch.Mismatch (A, B) = A'Length)
with
  Pre  => A'Length <= B'Length,
  Post => (Equal'Result =
             (A = B (B'First .. B'First - 1 + A'Length)));
```

Adding Option

An « option » type can be easily defined to avoid using length of the first array when the two arrays mismatch:

```
type Option is record
  Exists : Boolean;
  Value  : Natural;
end record;
```

Adding Option

Specification is straightforward:

```
function Mismatch (A : T_Arr; B : T_Arr) return Option with
  Pre => A'Length <= B'Length,
  Contract_Cases => (
    A = B (B'First .. B'First - 1 + A'Length) =>
      not Mismatch'Result.Exists,
    others                                      =>
      Mismatch'Result.Exists and then
      (A (A'First + Mismatch'Result.Value) /=
       B (B'First + Mismatch'Result.Value))
      and then
      (if (Mismatch'Result.Value /= 0) then
       Equal_Ranges.Equal_Ranges(A, B, Mismatch'Result.Value - 1)));
```

Adding Option

Implementation is immediate:

```
function Mismatch (A : T_Arr; B : T_Arr) return Option is
  Result : Option := (Exists => False, Value => 0);
begin
   for I in 0 .. A'Length - 1 loop
      if (A (A'First + I) /= B (B'First + I)) then
         Result.Exists := True;
         Result.Value  := I;

         return Result;
      end if;

      pragma Loop_Invariant
        (not Result.Exists);
      pragma Loop_Invariant
        (Equal_Ranges.Equal_Ranges (A, B, I));
      pragma Loop_Variant
        (Increases => I);
   end loop;

   return Result;
end Mismatch;
```

Conclusion on SPARK by Example

What has been done:

- Jérôme has already tackled first chapters from ACSL by Example 11.1, but in "C style"
- chapter on non-mutating algorithms is OK, but needs SPARK Pro 2017 as it uses array slices

What remains to do:

- "SPARKify" the current specifications/implementations
- add the new implementations from ACSL by Example 14.1
- do not hesitate to contribute to `https://github.com/tofgarion/spark_examples` (`GPL2016` or `PRO2017` branches)