Preuve auto-active de programmes en SPARK

Yannick Moy - AdaCore

La preuve pour le langage Ada

Ada - un langage fortement spécifié

- Résultat d'un concours du DoD américain
- Exigences raffinées de 1975 (*strawman*) à 1978 (*steelman*)
- Une définition formelle est demandée !
- Standard ISO en 1983 (puis 1995, 2005, 2012... 2022)

Plusieurs projets d'annotations logiques de programmes Ada

- ANNA - A Language for Annotating Ada Programs (1987)
- SPARK - the SPADE Ada Kernel (1987)
- Penelope - Formal Verification of Ada Programs (1990)

Le langage Ada pour la preuve

```
type Value is new Integer range -10_000 .. 10_000;

type Data is array (Positive range ◇) of Value;

procedure Sort (D : in out Data)
with
  Post ⇒ (for all I in D'Range ⇒
            (if I < D'Last then D(I) ≤ D(I+1)));
```

type différent

avec contrainte

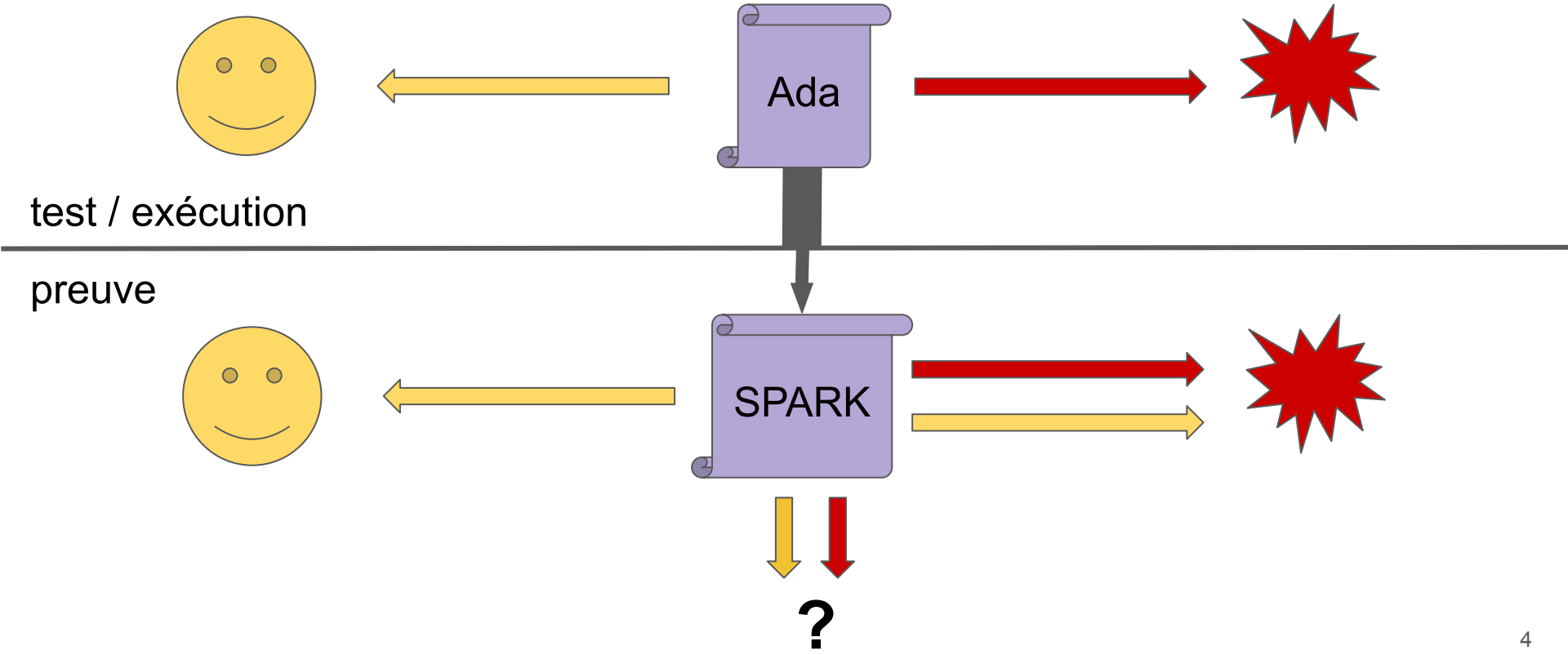
tableau avec indices positifs

paramètre d'entrée-sortie

postcondition

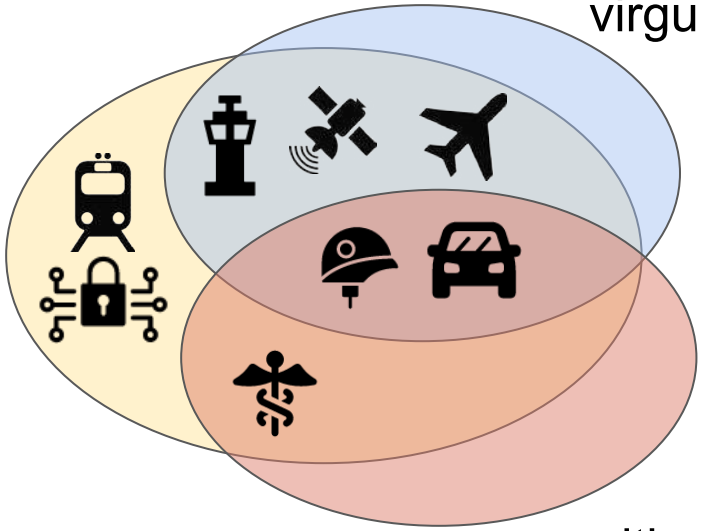
expressions riches : quantification, conditionnelle

Le sous-ensemble SPARK



Utilisation industrielle de SPARK

arithmétique à
virgule flottante

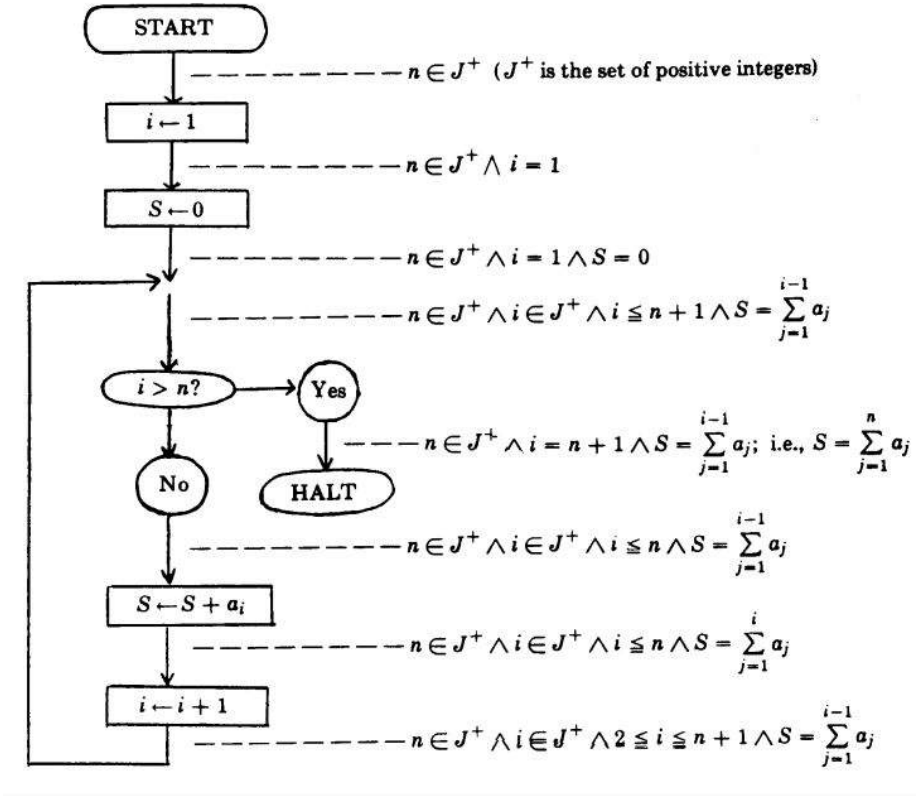


arithmétique
entière

arithmétique à
virgule fixe

cible : logiciels critiques
de taille intermédiaire
1k - 100k loc

Preuve d'algorithme ou de programme ?



Robert Floyd, 1967

Assigning Meaning to Programs

Leslie Lamport, 2018

If You're Not Writing a Program, Don't Use a Programming Language

Les spécifications exécutables : contraintes

```
Max_Value : constant := Integer'Last;
...
function Sum (A : Data; Up_To : Index) return Integer is
  (if Up_To = 0 then 0 else A(Up_To) + Sum (A, Up_To - 1));
```

tous les positifs

dépassements de capacité possibles

- problème de **correction** (« *soundness* »)
- impossible de contourner le problème avec des axiomes (absents de SPARK)
- même problème avec les réels mathématiques

Les spécifications exécutables : bénéfices

➢ compréhensibles par les ingénieurs

Patrice Chalin, *A Sound Assertion Semantics for the Dependable Systems Evolution Verifying Compiler*, 2007

➢ compréhensibles par les autres outils de développement

➢ on peut déboguer et tester les spécifications !

Bénéfices pour la **correction interne** et la **correction externe**

La preuve auto-active

automatique et inter**active**

Déjà le cas pour certaines spécifications :

- spécifications de sous-programmes internes
- invariants de boucle

Plus généralement utilisation de code « fantôme » sans effet visible

➢ assertions

➢ variables fantômes

➢ lemmes

Les preuves auto-actives

Prouver une propriété par induction

Fournir le témoin d'une propriété

```
pragma Assert (Property (This_Value));
```

Guider les prouveurs automatiques

```
pragma Assert (First_Fact);
pragma Assert (Second_Fact);
pragma Assert (Conclusion);
```

prouvé par A

prouvé par B

prouvé par C

Rendre la preuve automatique plus robuste

L’arithmétique machine

Entièrement décidable !

Impossible à décider en pratique

Difficultés avec les opérations non linéaires (multiplication, division, modulo)

```
pragma Assert (if B mod A = 0 and C mod B = 0 then C mod A = 0);
```

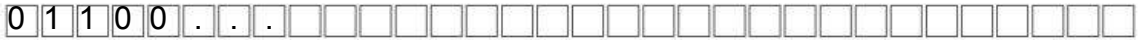
Difficultés avec les mélanges d’arithmétiques

```
pragma Assert (if A < 2**24 then Float(A) + 1.0 = Float(A + 1));
```

L’arithmétique machine : les entiers

Deux visions possibles des entiers machines (par exemple 32 bits)

mathématique : $-2^{31} \leq V \leq 2^{31} - 1$

informatique : 

Choix dans l’outil SPARK :

- encodage des entiers signés comme entiers mathématiques bornés
- encodage des entiers modulaires comme vecteurs de bits

➢ problèmes de combinaison des entiers signés et des entiers modulaires

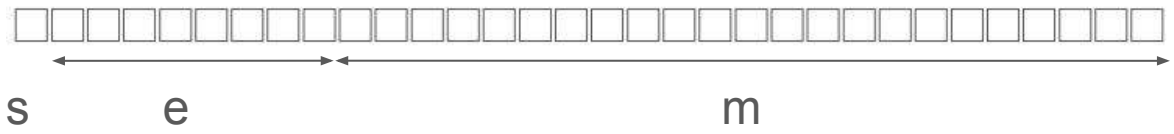
L’arithmétique machine : les réels

Réels en virgule fixe traités comme des entiers

Deux visions possibles des réels en virgule flottante (par exemple 32 bits)

mathématique : $V = (-1)^s \times m \times 2^e$; $0 \leq m < 2^{24}$; $-149 \leq e \leq 104$

informatique :



Choix dans les prouveurs :

- encodage des flottants comme réels bornés avec opération d’arrondi
- encodage des flottants comme vecteurs de bits

➢ chaque encodage est meilleur pour certains types de problèmes

Vérification modulaire - sous-programmes et boucles

```
function Sum_SPARK (A : Data) return Integer is
   S : Integer := 0;
begin
   for I in A'Range loop
      S := S + A(I);
      pragma Loop_Invariant (S = Sum (A, I));
   end loop;
   return S;
end Sum_SPARK;
```

besoin de résumer le contexte

besoin de résumer les effets de la fonction

besoin de résumer les effets de l'itération

La condition-cadre - valeurs invariantes

```
  procedure Sum_SPARK (A : in out Data_Sum)
  with
    Post ⇒ A.S = Sum (A.D)
      and then A.D = A.D'Old;
…
       for I in A.D'Range loop
           A.S := A.S + A.D(I);
           pragma Loop_Invariant (A.S = Sum (A.D, I));
           pragma Loop_Invariant (A.D = A.D'Loop_Entry);
       end loop;
```

champ D non modifié par l'appel

champ D non modifié par la boucle

Automatisation et interaction

Problème bien connu de l’automatisation : elle échoue dans les cas difficiles

Chaque automatisation des cas simples rend les cas difficiles plus difficiles

➢ solution : la **preuve auto-active**

Problème « classique » de découverte de l’invariant de boucle

➢ pas le problème principal en pratique

➢ catalogue d’invariants de boucles + méthodologie d’écriture auto-active

Investigation des échecs de preuve

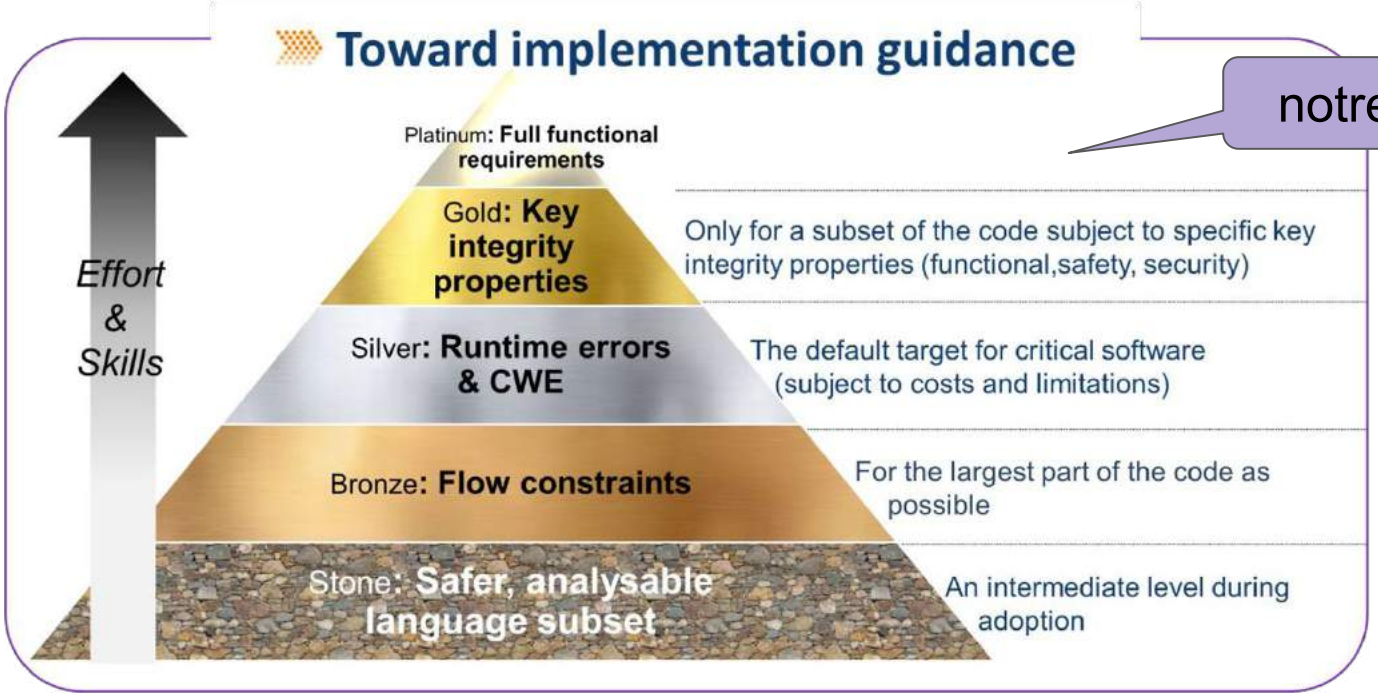
Raisons possibles de l'échec :

- code incorrect
- spécification incorrecte
- spécification incomplète (condition-cadre)
- limitation du modèle dans l'outil SPARK
- le prouveur a besoin de plus de temps
- limitation intrinsèque des prouveurs (induction, arithmétique non linéaire)

➢ messages de l'outil essentiels

➢ preuve auto-active jusqu'à succès ou échec de preuve élémentaire

Des niveaux de garanties

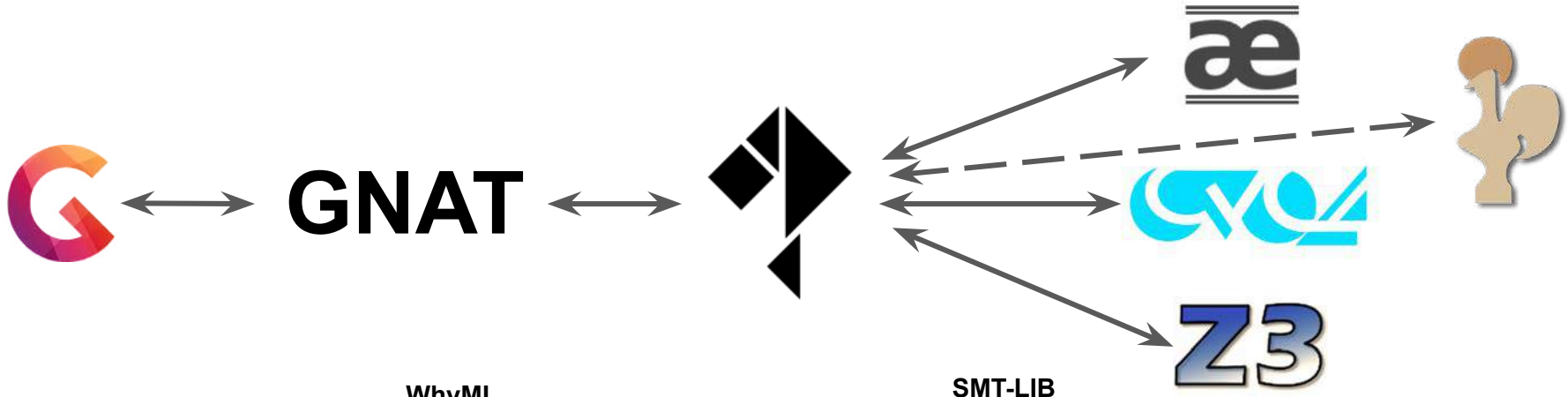


notre exemple `Sum`

aligné avec les niveaux d'assurance en certification : DAL, SIL

https://www.adacore.com/papers/climbing-the-software-assurance-ladder

Un outil de l'écosystème du logiciel libre



SPARK

```
A(1) := 42;
```

WhyML

```
a.map__content <-
  set
    (a.map__content)
    (let temp = 1 : int in
       assert { temp ... };
       temp)
    (42 : value)
```

SMT-LIB

```
(assert
  (not
  (=> (dynamic_property 0 1000000
         (to_rep a__first) (to_rep a__last))
  (=> (and (= (to_rep a__first) 1)
           (<= 0 (to_rep a__last)))
  (<= (to_rep a__first) 1)))))
(check-sat)
```

Preuve d’initialisation

Politique d’initialisation par défaut : entrées-sorties entièrement initialisées

→ analyse statique de flux de données

Relâchement explicite de la politique d’initialisation → preuve

```
type Data is array (Index range <>) of Value with
  Relaxed_Initialization;
```

Spécification explicite de l’initialisation en pré / post / invariant de boucle :

```
(for all J in 1 .. S ⇒ A(J)'Initialized)
```

Preuve de non-interférence

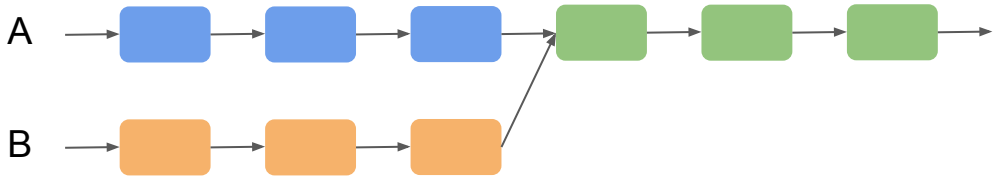
Pas de partage mémoire (*aliasing*) entre entrées-sorties en interférence
- lecture/écriture, ou
- écriture/écriture

→ analyse statique de flux de données

Programmes sans pointeurs : comparaison des chemins d'accès

➢ partage mémoire entre X et X ; X et X.C ; X(J) et X(K)
➢ pas de partage mémoire entre X et Y ; X.A et X.B ; X(1) et X(2)

Le modèle des pointeurs



Choix dans SPARK : pas de partage mémoire en interférence entre zones mémoires pointées

➤ pas de « modèle mémoire »

➤ politique d'appartenance (*ownership*) des zones pointées

➤ nouvelles vérifications : pointeur valide, absence de fuite mémoire

Emprunt (*borrow*), promesse (*pledge*) et valeur prophétique

`P` emprunte la mémoire pointée par `A` dans `Init_List`

`At_End(A)` dénote la valeur *prophétique constante* de `A` à la fin de l'emprunt…
`At_End(P)` dénote la valeur *prophétique variable* de `P` à la fin de l'emprunt…

sans pouvoir prédire quelles seront ces valeurs !

L'invariant de boucle exprime la promesse maintenue inductivement :

```
Sum (At_End (A)) = Sum (At_End (P))
```

Astrauskas, Müller, Poli, Summers - *Leveraging Rust Types for Modular Specification and Verification*
Dross, Kanig - *Recursive Data Structures in SPARK*
Matsushita, Tsukada, Kobayashi - *RustHorn: CHC-based Verification for Rust Programs*

Pour terminer

Terminaison prouvée pour `Sum` (axiome généré essentiel à la preuve)

Vérification en SPARK correcte pour les programmes qui ne terminent pas
➤ garanties de la preuve en chaque point de programme

Spécification de **terminaison** → analyse de flux de données et preuve

Spécification de **non-terminaison** → analyse de flux de données et preuve

Spécification de **possible non-terminaison**
➤ procédures dans lesquelles il est correct d'arrêter l'exécution
➤ contraintes spécifiques d'appel à ces procédures → analyse de flux de données

Conclusion

Preuve de programme utilisable par des ingénieurs

Nécessité de restreindre le langage de programmation

Utilisabilité repose sur le binôme automatisation - interaction

Acceptabilité dépend de facteurs externes : normes applicables, conséquences des erreurs, cycles d'adoption de technologies

Pour aller plus loin : **learn.adacore.com** et **www.adacore.com/download**

Exemples présentés ici : **github.com/yannickmoy/am2sp/**