

Vérification déductive de programmes avec Why3

Jean-Christophe Filliâtre
CNRS

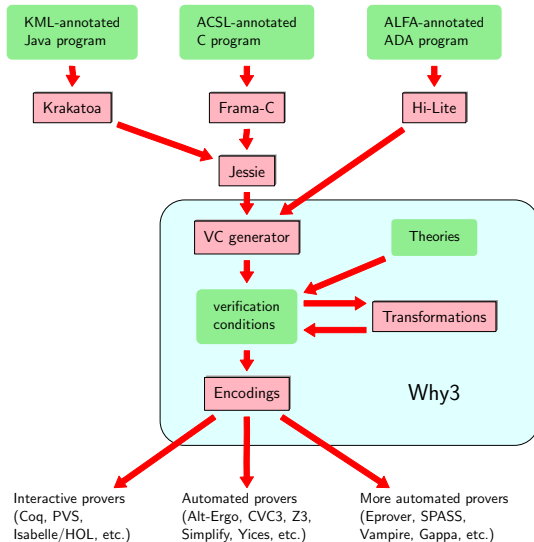
Journées sur la Fiabilité du Logiciel
Paris, 14 juin 2012

- un outil pour la vérification déductive de programmes

programme+spécif. $\longrightarrow$ formule logique $\longrightarrow$ preuve

- rendu possible par la révolution SMT

- développé depuis 2001 dans l'équipe ProVal (LRI / INRIA)
- complètement réécrit en 2010
(F. Bobot, JCF, C. Marché, G. Melquiond, A. Paskevich)
- logiciel libre (LGPL)
<http://why3.lri.fr/>
- un outil similaire : Boogie (Microsoft Research)



parler à l'oreille des démonstrateurs

- démonstrateurs interactifs
 - logique riche : polymorphisme, ordre supérieur, types dépendants, types algébriques, etc.
- démonstrateurs automatiques
 - logique plus pauvre : premier ordre, types simples, etc.
 - théories prédéfinies : arithmétique, tableaux, types énumérés, etc.

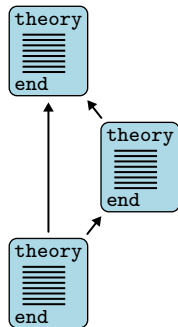
Why3 définit une logique de compromis

- logique du premier ordre
- polymorphisme
- types algébriques
- définitions récursives
- prédicats inductifs

déclarations logiques organisées
en petites théories

idées

- contrôler le contexte logique
- théories paramétriques réutilisables

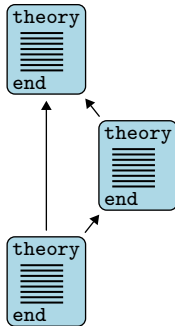


le concept central est celui de tâche

- un contexte = une liste de déclarations
- un but = une formule



parcours d'une tâche

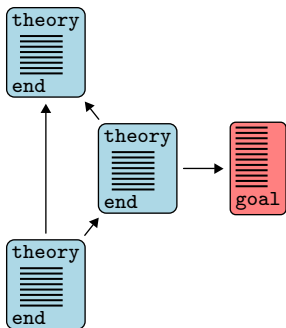


Alt-Ergo

Z3

Vampire

parcours d'une tâche

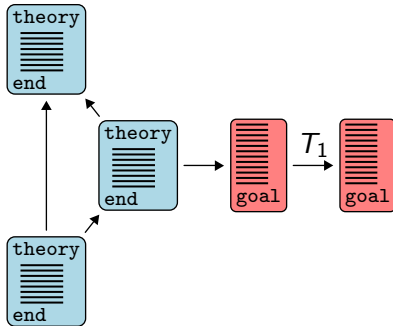


Alt-Ergo

Z3

Vampire

parcours d'une tâche

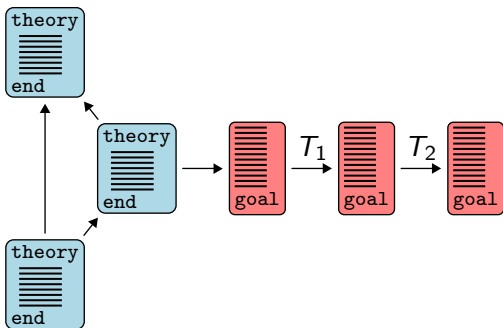


Alt-Ergo

Z3

Vampire

parcours d'une tâche

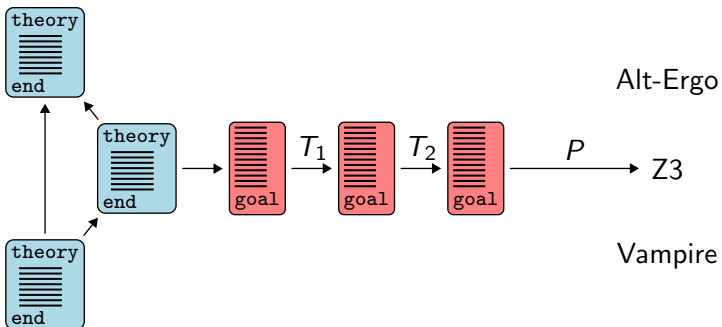


Alt-Ergo

Z3

Vampire

parcours d'une tâche



le parcours d'une tâche est piloté par un fichier

- transformations à appliquer
- format d'entrée du démonstrateur
 - syntaxe
 - symboles prédéfinis
- diagnostics des messages du démonstrateur

Exemple : pilote Z3 (extrait)

```
printer "smtv2"
valid "^unsat"
invalid "^sat"

transformation "inline_trivial"
transformation "eliminate_builtin"
transformation "eliminate_definition"
transformation "eliminate_inductive"
transformation "eliminate_algebraic"
transformation "simplify_formula"
transformation "discriminate"
transformation "encoding_smt"

prelude "(set-logic AUFNIRA)"

theory BuiltIn
  syntax type int "Int"
  syntax type real "Real"
  syntax predicate (=) "(= %1 %2)"

  meta "encoding : kept" type int
end
```

Why3 offre une API OCaml

- pour construire des formules, des théories, des tâches
- pour appeler des démonstrateurs

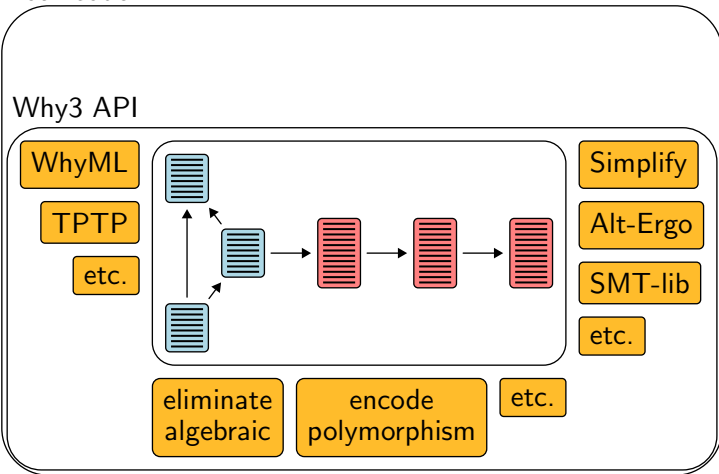
API **défensive**

- formules, théories, tâches bien formées

Why3 peut être étendu par trois sortes de greffons

- langages d'entrée
- transformations (utilisées dans les pilotes)
- syntaxe de sortie (pour les démonstrateurs)

Your code



une preuve de programme

N personnes votent chacune pour un candidat

A	A	A	C	C	B	B	C	C	C	B	C	C
---	---	---	---	---	---	---	---	---	---	---	---	---

on veut déterminer si un candidat obtient la majorité absolue
(et, le cas échéant, le renvoyer)

due à Boyer & Moore (1980)

en temps linéaire

utilise seulement trois variables

MJRTY—A Fast Majority Vote Algorithm¹

Robert S. Boyer and J Strother Moore

Computer Sciences Department
University of Texas at Austin
and
Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas

Abstract

A new algorithm is presented for determining which, if any, of an arbitrary number of candidates has received a majority of the votes cast in an election.

A	A	A	C	C	B	B	C	C	C	B	C	C
---	---	---	---	---	---	---	---	---	---	---	---	---



cand = A

k = 1

A	A	A	C	C	B	B	C	C	C	B	C	C
---	---	---	---	---	---	---	---	---	---	---	---	---



cand = A

k = 2

A	A	A	C	C	B	B	C	C	C	B	C	C
---	---	---	---	---	---	---	---	---	---	---	---	---



cand = A

k = 3

A	A	A	C	C	B	B	C	C	C	B	C	C
---	---	---	---	---	---	---	---	---	---	---	---	---



cand = A

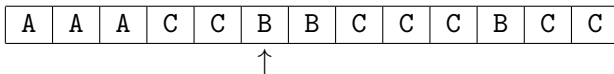
k = 2

A	A	A	C	C	B	B	C	C	C	B	C	C
---	---	---	---	---	---	---	---	---	---	---	---	---



cand = A

k = 1



cand = A

k = 0

A	A	A	C	C	B	B	C	C	C	B	C	C
---	---	---	---	---	---	---	---	---	---	---	---	---



cand = B

k = 1

A	A	A	C	C	B	B	C	C	C	B	C	C
---	---	---	---	---	---	---	---	---	---	---	---	---



cand = B

k = 0

A	A	A	C	C	B	B	C	C	C	B	C	C
---	---	---	---	---	---	---	---	---	---	---	---	---



cand = C

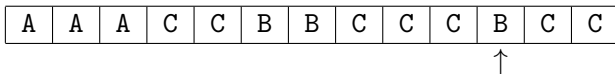
k = 1

A	A	A	C	C	B	B	C	C	C	B	C	C
---	---	---	---	---	---	---	---	---	---	---	---	---



cand = C

k = 2



cand = C

k = 1

A	A	A	C	C	B	B	C	C	C	B	C	C
---	---	---	---	---	---	---	---	---	---	---	---	---



cand = C

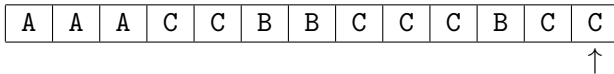
k = 2

A	A	A	C	C	B	B	C	C	C	B	C	C
---	---	---	---	---	---	---	---	---	---	---	---	---



cand = C

k = 3



`cand = C`

`k = 3`

on vérifie alors si C a bien la majorité absolue (ici c'est le cas)

```

SUBROUTINE MJRTY(A, N, BOOLE, CAND)
  INTEGER N
  INTEGER A
  LOGICAL BOOLE
  INTEGER CAND
  INTEGER I
  INTEGER K
  DIMENSION A(N)
  K = 0
C   THE FOLLOWING DO IMPLEMENTS THE PAIRING PHASE. CAND IS
C   THE CURRENTLY LEADING CANDIDATE AND K IS THE NUMBER OF
C   UNPAIRED VOTES FOR CAND.
  DO 100 I = 1, N
    IF ((K .EQ. 0)) GOTO 50
    IF ((CAND .EQ. A(I))) GOTO 75
    K = (K - 1)
    GOTO 100
50   CAND = A(I)
    K = 1
    GOTO 100
75   K = (K + 1)
100  CONTINUE
    IF ((K .EQ. 0)) GOTO 300
    BOOLE = .TRUE.
    IF ((K .GT. (N / 2))) RETURN
C   WE NOW ENTER THE COUNTING PHASE. BOOLE IS SET TO TRUE
C   IN ANTICIPATION OF FINDING CAND IN THE MAJORITY. K IS
C   USED AS THE RUNNING TALLY FOR CAND. WE EXIT AS SOON
C   AS K EXCEEDS N/2.
    K = 0
    DO 200 I = 1, N
      IF ((CAND .NE. A(I))) GOTO 200
      K = (K + 1)
      IF ((K .GT. (N / 2))) RETURN
200  CONTINUE
300  BOOLE = .FALSE.
    RETURN
  END

```

```
let mjrty (a: array candidate) =  
  let n = length a in  
  let cand = ref a[0] in let k = ref 0 in  
  for i = 0 to n-1 do  
    if !k = 0 then begin cand := a[i]; k := 1 end  
    else if !cand = a[i] then incr k else decr k  
  done;  
  if !k = 0 then raise Not_found;  
  try  
    if 2 * !k > n then raise Found; k := 0;  
    for i = 0 to n-1 do  
      if a[i] = !cand then begin  
        incr k; if 2 * !k > n then raise Found  
      end  
    done;  
    raise Not_found  
  with Found →  
    !cand  
end
```

- précondition

```
let mjrty (a: array candidate) =
  { 1 ≤ length a }
  ...
```

- postcondition en cas de succès

```
...
{ 2 * numof a result 0 (length a) > length a }
```

- postcondition en cas d'échec

```
| Not_found →
{ ∀ c: candidate.
  2 * numof a c 0 (length a) ≤ length a }
```

chaque boucle est munie d'un **invariant**

```

for i = 0 to n-1 do
  invariant {  $0 \leq !k \leq i \wedge$ 
    numof a !cand 0 i  $\geq$  !k  $\wedge$ 
     $2 * (\text{numof a !cand 0 i} - !k) \leq i - !k \wedge$ 
     $\forall c: \text{candidate.}$ 
     $c \neq !cand \rightarrow 2 * \text{numof a c 0 i} \leq i - !k$ 
  }
  ...

for i = 0 to n-1 do
  invariant {  $!k = \text{numof a !cand 0 i} \wedge 2 * !k \leq n$  }
  ...

```

l'obligation de preuve exprime

- la sûreté d'exécution
 - accès dans les bornes
 - terminaison
- le respect des annotations
 - invariants établis initialement et préservés
 - postconditions vérifiées

elle est entièrement prouvée par Alt-Ergo

Why3, c'est

- un outil pour parler aux démonstrateurs
- un langage
 - pour prouver des algorithmes
 - intermédiaire pour la preuve de programmes
- un logiciel libre

<http://why3.lri.fr/>

merci