Vérification déductive de programmes

Jean-Christophe Filliâtre
CNRS

Acquérir une culture commune dans le domaine de l'informatique

lycée Diderot
30 mai 2017

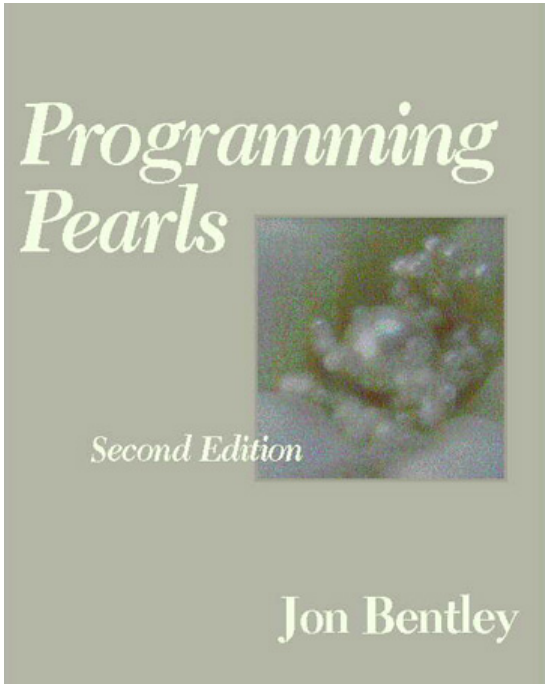
Software is hard. – Don Knuth

pourquoi ?

- mauvaise interprétation des spécifications
- programmation dans l'urgence
- changements incompatibles
- logiciel = objet très complexe
- etc.

un exemple célèbre : *binary search*

première publication en 1946
première publication sans bug en 1962



Jon Bentley. Programming Pearls. 1986.

Writing correct programs

the challenge of binary search

et pourtant...

en 2006, un bug a été trouvé dans le code de *binary search* de la bibliothèque standard de Java

> Joshua Bloch, Google Research Blog
> *"Nearly All Binary Searches and Mergesorts are Broken"*

ce bug était là depuis 9 ans

```
...
int mid = (low + high) / 2;
int midVal = a[mid];
...
```

peut provoquer un débordement de capacité arithmétique,
suivi d'un accès en dehors des bornes du tableau

un correctif possible

```
int mid = low + (high - low) / 2;
```

de meilleurs langages de programmation

- meilleure syntaxe
  (éviter de considérer `DO 17 I = 1. 10` comme une affectation)

- plus de typage
  (éviter de confondre des mètres et des yards)

- plus d'avertissements du compilateur
  (éviter d'oublier certains cas)

- etc.

le test systématique et rigoureux est une autre réponse, complémentaire

mais le test est

- coûteux
- parfois très difficile à mettre en œuvre
- et surtout incomplet (à de très rares exceptions près)

les méthodes formelles proposent une approche mathématique de la correction du logiciel

qu'est-ce qu'un programme ?

il y a plusieurs aspects en jeux

- ce que l'on calcule (quoi)
- la manière de le calculer (comment)
- la raison pour laquelle c'est correct (pourquoi)

qu'est-ce qu'un programme ?

le programme, ce n'est que le « comment », et rien d'autre

le « quoi » et le « pourquoi » n'en font pas partie

ce sont des cahiers des charges, des commentaires, des pages web, des croquis, des articles de recherche, etc.

- comment : 2 lignes de C

```
a[52514],b,c=52514,d,e,f=1e4,g,h;main(){for(;b=c-=14;h=printf("%04d",
e+d/f))for(e=d%=f;g=--b*2;d/=g)d=d*b+f*(h?a[b]:f/5),a[b]=d%--g;}
```

- comment : 2 lignes de C

```
a[52514],b,c=52514,d,e,f=1e4,g,h;main(){for(;b=c-=14;h=printf("%04d",
e+d/f))for(e=d%=f;g=--b*2;d/=g)d=d*b+f*(h?a[b]:f/5),a[b]=d%--g;}
```

- quoi : 15 000 décimales de π

- pourquoi : beaucoup de maths, dont

$$\pi = \sum_{i=0}^{\infty} \frac{(i!)^2\, 2^{i+1}}{(2i+1)!}$$

les méthodes formelles proposent une approche rigoureuse de la programmation, où on se donne

- une spécification écrite dans un langage mathématique
- une preuve que le programme vérifie cette spécification

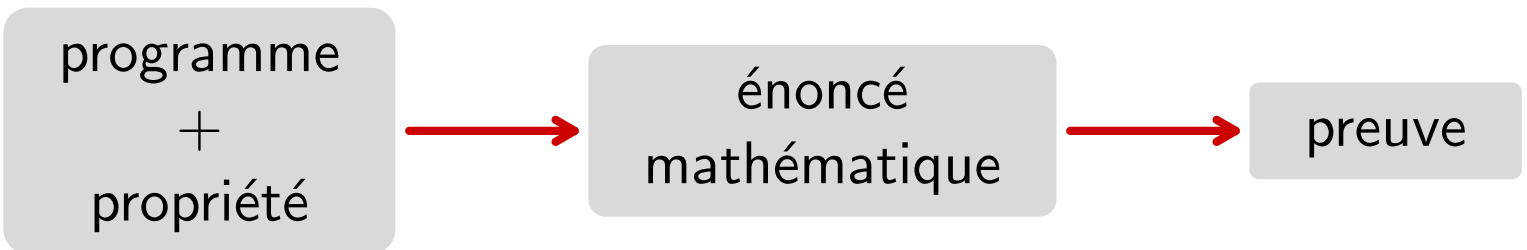
que souhaite-t-on prouver ?

- sûreté : le programme ne ≪ plante ≫ pas
  - pas d'accès illégal à la mémoire
  - pas d'opération illégale, comme une division par zéro
  - le programme termine

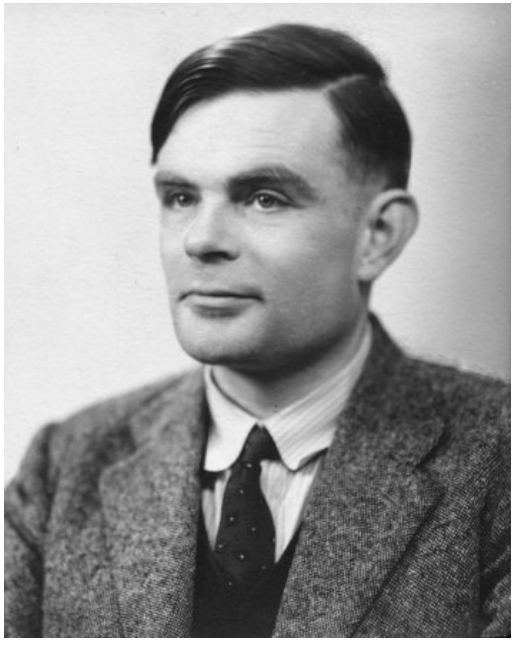
- correction fonctionnelle
  - le programme fait ce qu'il est censé faire

on peut citer le model checking, l'interprétation abstraite, etc.

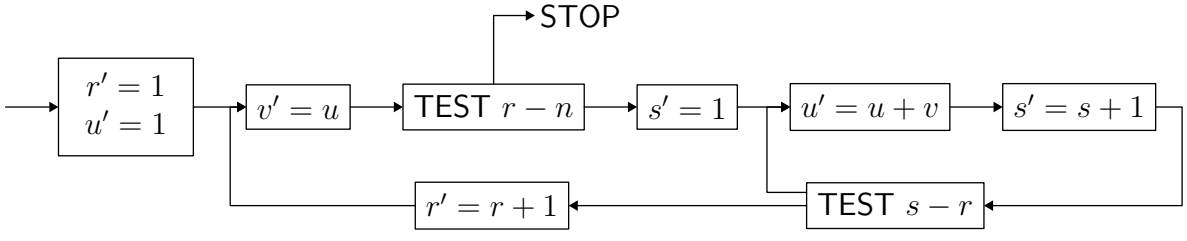
cet exposé présente la vérification déductive

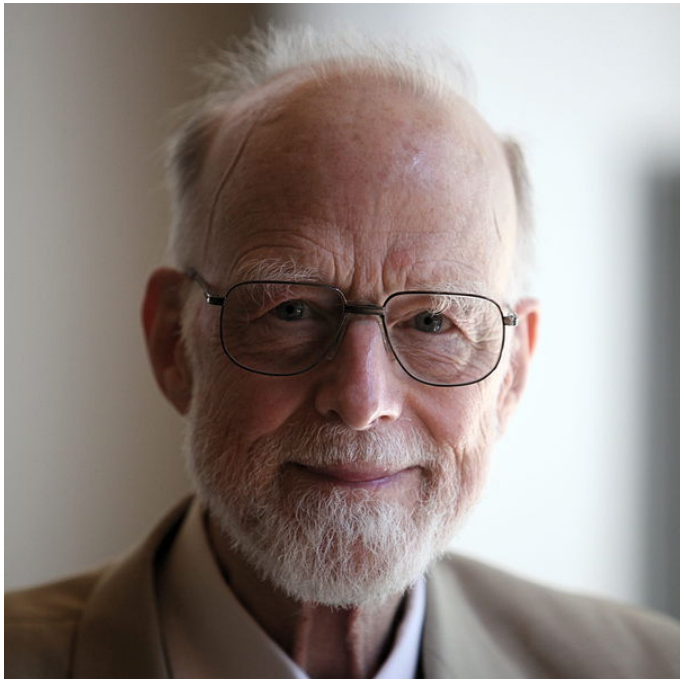


ce n'est pas nouveau

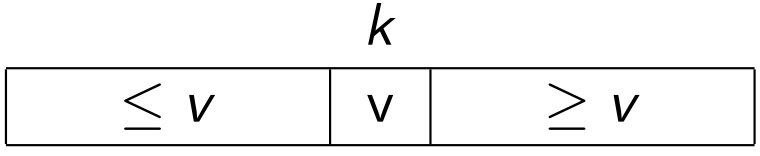


A. M. Turing. Checking a large routine. 1949.

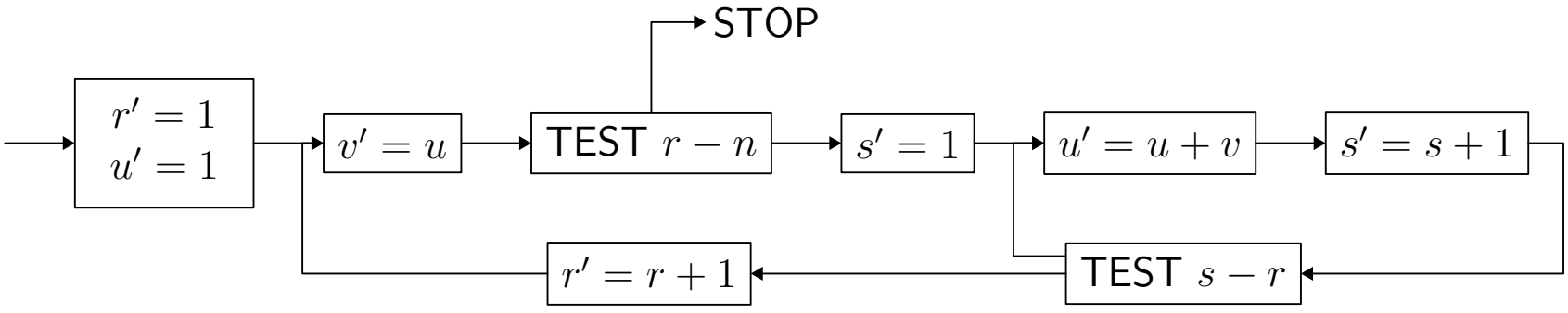




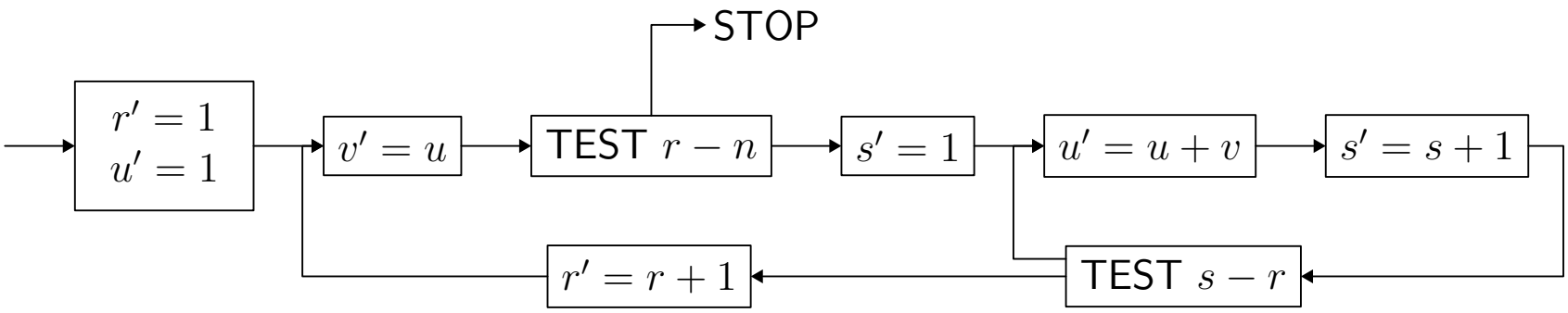
Tony Hoare.
Proof of a program : FIND.
Commun. ACM, 1971.



checking a large routine (Turing, 1949)

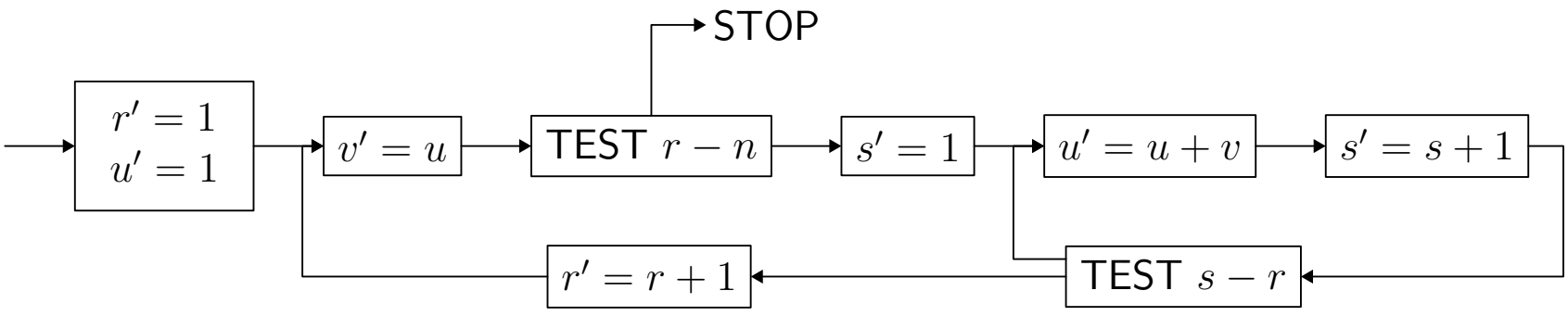


checking a large routine (Turing, 1949)



$u \leftarrow 1$
for $r = 0$ **to** $n - 1$ **do**
  $v \leftarrow u$
  for $s = 1$ **to** r **do**
    $u \leftarrow u + v$

checking a large routine (Turing, 1949)



précondition $\{n \geq 0\}$

$u \leftarrow 1$

for $r = 0$ **to** $n - 1$ **do**

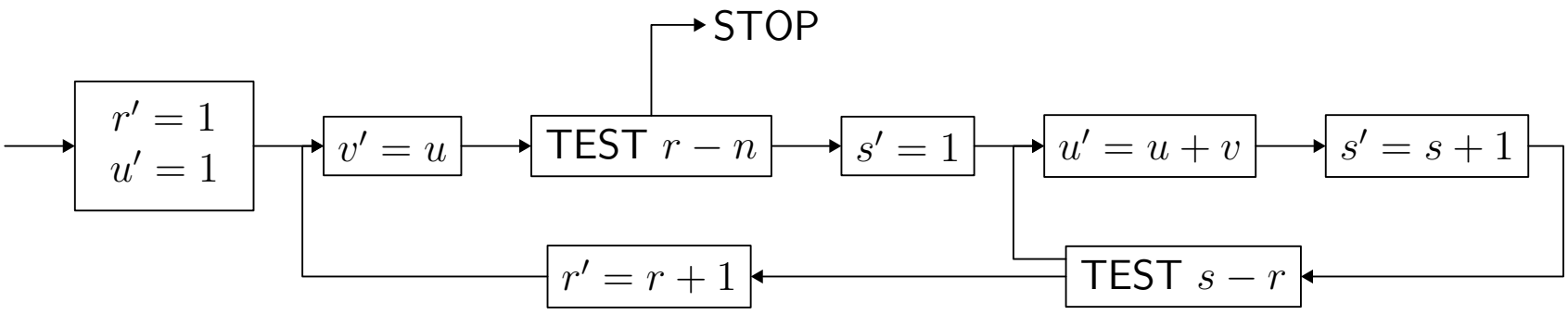
  $v \leftarrow u$

  for $s = 1$ **to** r **do**

    $u \leftarrow u + v$

postcondition $\{u = \mathit{fact}(n)\}$

checking a large routine (Turing, 1949)



précondition $\{n \geq 0\}$

$u \leftarrow 1$

for $r = 0$ **to** $n - 1$ **do** invariant $\{u = \mathit{fact}(r)\}$

  $v \leftarrow u$

  for $s = 1$ **to** r **do** invariant $\{u = s \times \mathit{fact}(r)\}$

    $u \leftarrow u + v$

postcondition $\{u = \mathit{fact}(n)\}$

```
function fact(int) : int
axiom fact0: fact(0) = 1
axiom factn: forall n:int. n >= 1 -> fact(n) = n * fact(n-1)

----------------------------------------------------------------------
goal vc: forall n:int. n >= 0 ->
  (0 > n - 1 -> 1 = fact(n)) /\
  (0 <= n - 1 ->
    1 = fact(0) /\
    (forall u:int.
      (forall r:int. 0 <= r /\ r <= n - 1 -> u = fact(r) ->
        (1 > r -> u = fact(r + 1)) /\
        (1 <= r ->
          u = 1 * fact(r) /\
          (forall u1:int.
            (forall s:int. 1 <= s /\ s <= r -> u1 = s * fact(r)
              (forall u2:int.
                 u2 = u1 + u -> u2 = (s + 1) * fact(r))) /\
            (u1 = (r + 1) * fact(r) -> u1 = fact(r + 1))))) /\
      (u = fact((n - 1) + 1) -> u = fact(n))))
```

condition de vérification

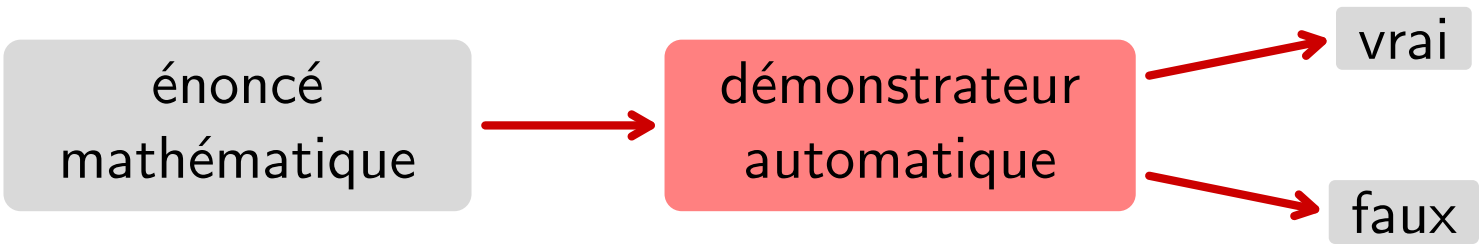
```
function fact(int) : int
axiom fact0: fact(0) = 1

----------------------------------------------------------------------
goal vc: forall n:int. n >= 0 ->
  (0 > n - 1 -> 1 = fact(n)) /\
```

que faire de cet énoncé mathématique ?

bien sûr, on pourrait le prouver à la main (comme Turing et Hoare)
mais c'est long, fastidieux, sujet à de nombreuses erreurs

aussi, on se tourne vers des outils qui mécanisent le raisonnement mathématique

démonstration automatique

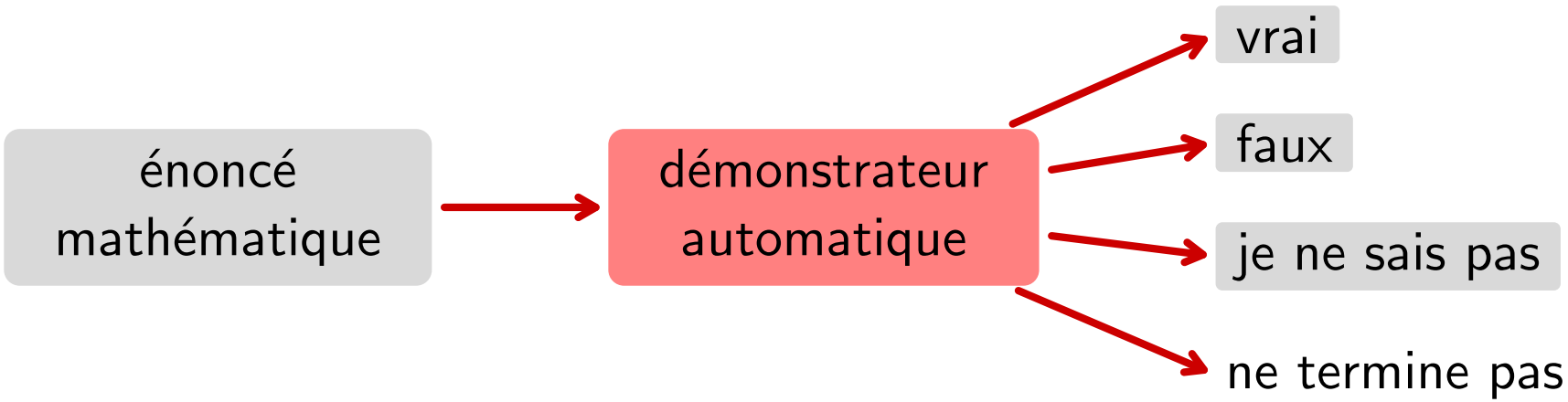


il n'est pas possible d'écrire un tel programme
(Turing/Church, 1936, d'après Gödel)

c'est le théorème anti-chômage pour les mathématiciens

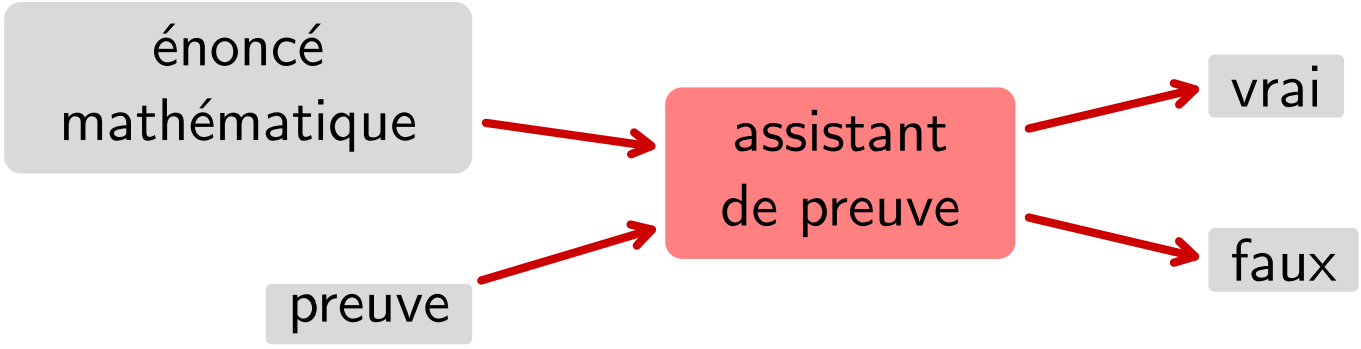
Kurt Gödel

démonstration automatique



exemples : Z3, CVC4, Alt-Ergo, Vampire, SPASS, etc.

si on se contente de vérifier une preuve, cela redevient décidable



exemples : Coq, Isabelle, PVS, HOL-light, etc.

Why3, un outil de vérification déductive

idée centrale : utiliser le plus grand nombre possible de démonstrateurs, tant automatiques qu'interactifs

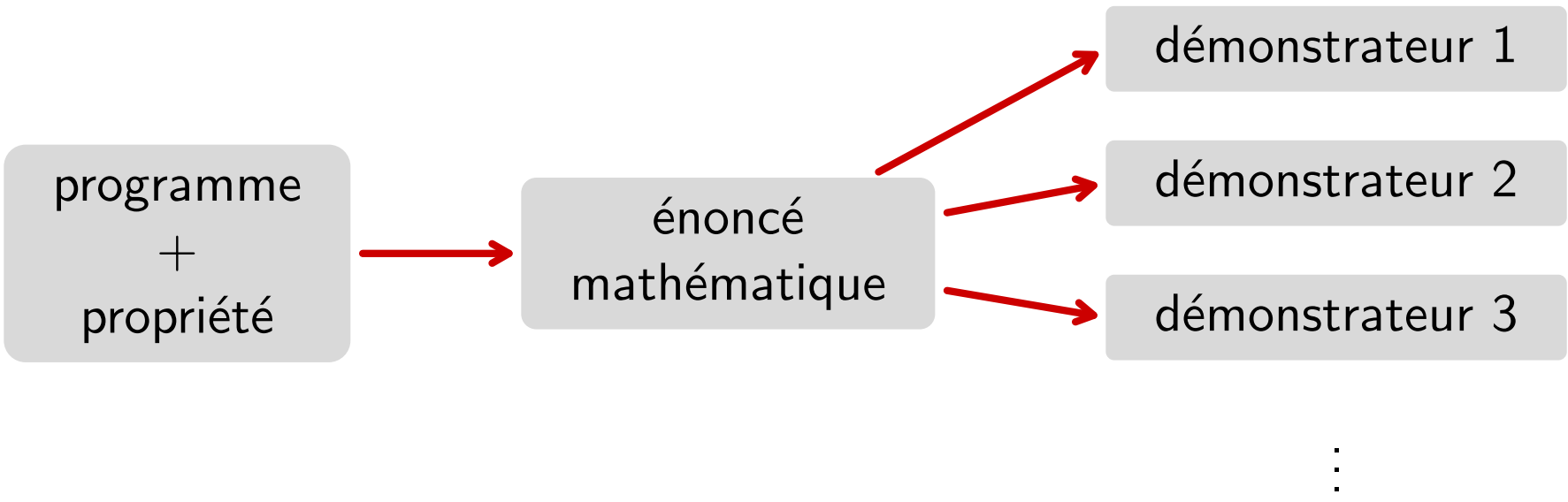
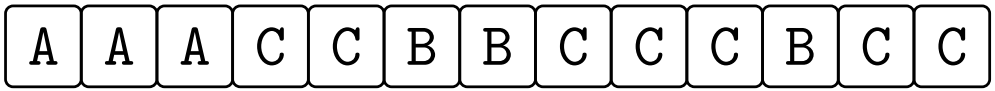


illustration sur un exemple

un ensemble de N votes est donné



déterminer si un candidat obtient la majorité absolue

une solution élégante

due à Boyer & Moore (1980)

en temps linéaire

en espace constant

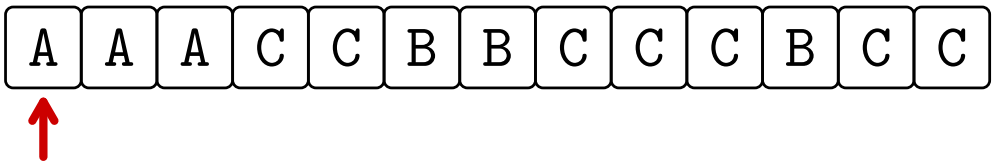
MJRTY—A Fast Majority Vote Algorithm[1]

Robert S. Boyer and J Strother Moore

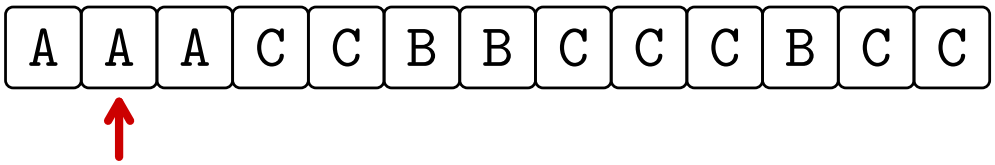
Computer Sciences Department
University of Texas at Austin
and
Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas

Abstract

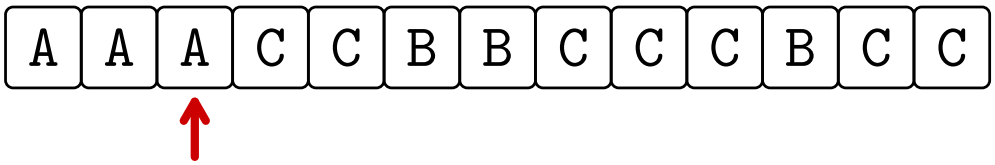
A new algorithm is presented for determining which, if any, of an arbitrary number of candidates has received a majority of the votes cast in an election.



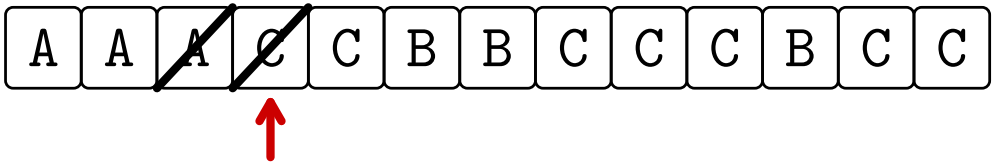

```
cand = A
k    = 1
```



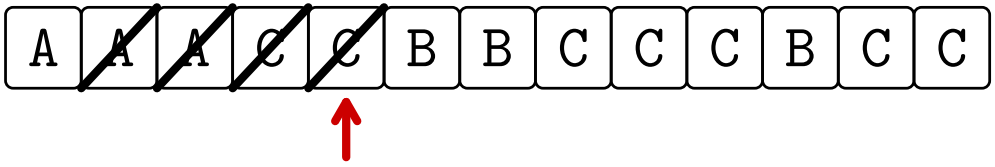

```
cand = A
k    = 2
```



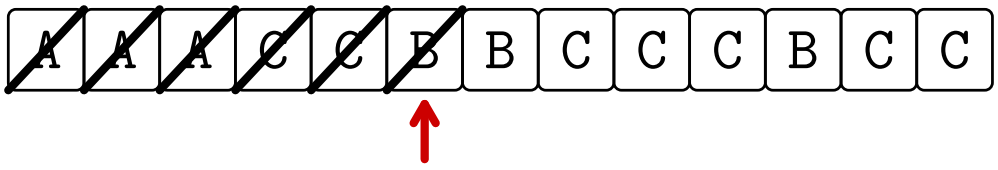

```
cand = A
k    = 3
```



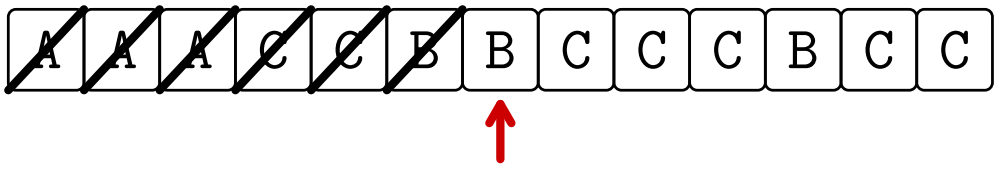

```
cand = A
k    = 2
```



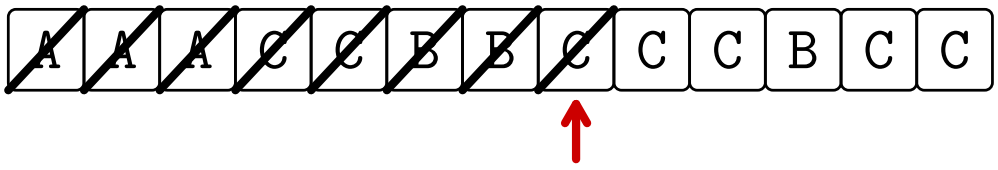

```
cand = A
k    = 1
```



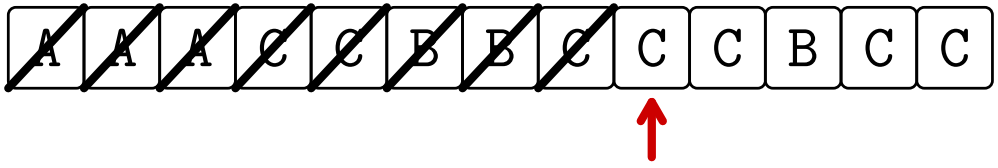

```
cand = A
k    = 0
```



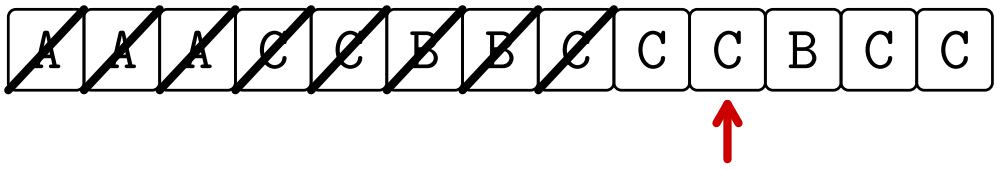

```
cand = B
k    = 1
```



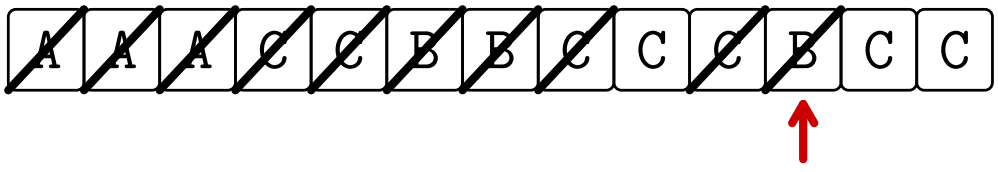

```
cand = B
k    = 0
```



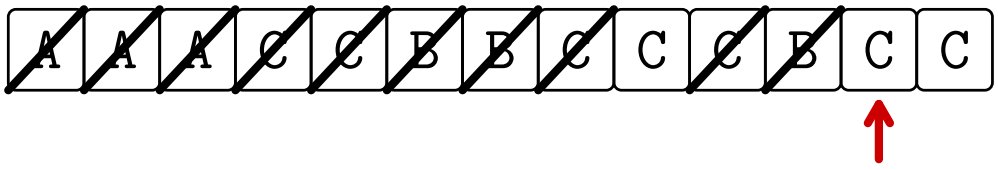

```
cand = C
k    = 1
```



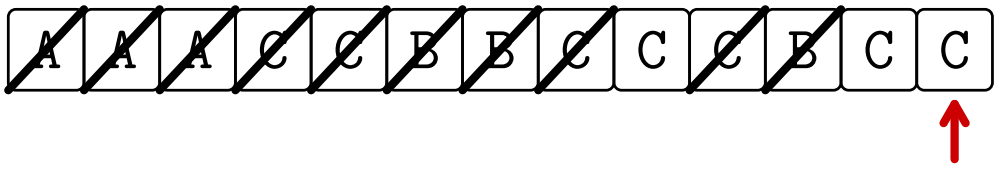

```
cand = C
k    = 2
```



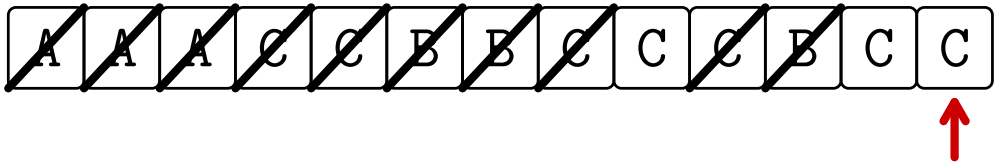

```
cand = C
k    = 1
```




```
cand = C
k    = 2
```




```
cand = C
k    = 3
```




```
cand = C
k    = 3
```

puis on vérifie si C a effectivement la majorité,
avec une seconde passe
dans ce cas, c'est vrai : $7 > 13/2$

code en Fortran

```
      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
```

code en Why3

```
let mjrty (a: array candidate) : 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
```

prouvons ce programme

- précondition

```
let mjrty (a: array candidate)
  requires { 1 <= length a }
```

- postcondition en cas de succès

```
  ensures
    { 2 * numeq a result 0 (length a) > length a }
```

- postcondition en cas d'échec

```
  raises { Not_found ->
           forall c: candidate.
           2 * numeq a c 0 (length a) <= length a }
```

première boucle

```
for i = 0 to n - 1 do
  invariant { 0 <= !k <= numeq a !cand 0 i }
  invariant { 2 * (numeq a !cand 0 i - !k) <= i - !k }
  invariant { forall c: candidate. c <> !cand ->
                2 * numeq a c 0 i <= i - !k  }
  ...
```

seconde boucle

```
for i = 0 to n - 1 do
  invariant { !k = numeq a !cand 0 i }
  invariant { 2 * !k <= n }
  ...
```

la condition de vérification exprime

- la sûreté
  - accès dans les bornes du tableau
  - terminaison
- respect des spécifications
  - les invariants sont initialisés et préservés
  - les postconditions sont établies

elle est entièrement prouvée par un démonstrateur automatique

plus de détails sur

http://why3.lri.fr/

- logiciel libre
- une centaine de programmes prouvés
- documentation, notes de cours (y compris en français)

- la vérification déductive est une méthode formelle de preuve de programme (ce n'est pas la seule)
- elle s'appuie en particulier sur les démonstrateurs, automatiques et interactifs, qui mécanisent les raisonnements logiques
- cela reste un processus très coûteux, notamment en moyens humains (écrire des spécifications, des invariants, des preuves)

défis scientifiques

- définir de meilleurs langages de programmation, mieux adaptés à la preuve
- définir de meilleurs langages logiques, plus expressifs
- définir de meilleurs démonstrateurs automatiques

} tension

des questions ?