Prouver les programmes
Pourquoi, quand, comment

Gérard Berry

Collège de France
Chaire Algorithmes, machines et langages
gerard.berry@college-de-france.fr

Cours 2, Paris, 04/03/2015

Suivi du séminaire de Joseph Sifakis
Prix Turing 2007

COLLÈGE
DE FRANCE
1530

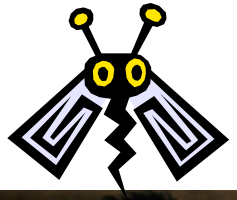
Agenda

1. Le bug, un danger de l'informatique
2. Comment réduire les bugs
3. Les méthodes de test : forces et faiblesses
4. La naissance de la vérification formelle
5. Les méthodes de vérification formelles
   1. Les assertions
   2. La réécriture symbolique
   3. La sémantique dénotationnelle
   4. Les logiques et assistants de preuve
   5. Le model-checking
6. Etat et perspectives

Agenda

1. Le bug, un danger de l'informatique
2. Comment réduire les bugs
3. Les méthodes de test : forces et faiblesses
4. La naissance de la vérification formelle
5. Les méthodes de vérification formelles
   1. Les assertions
   2. La réécriture symbolique
   3. La sémantique dénotationnelle
   4. Les logiques et assistants de preuve
   5. Le model-checking
6. Etat et perspectives

Homme / ordinateur, un gouffre à combler

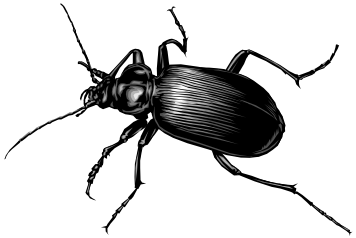


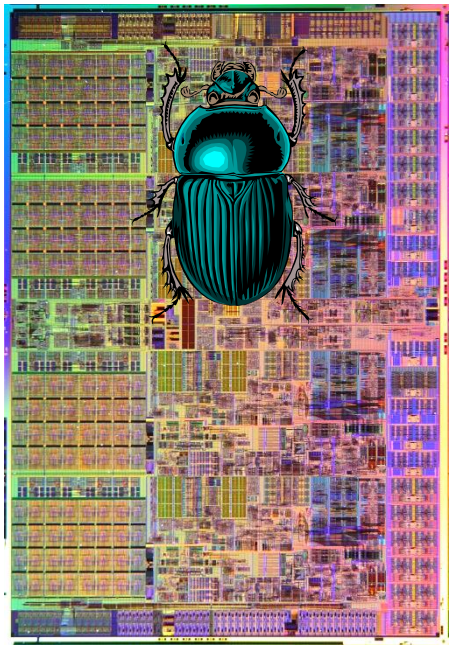
Intuition

Rigueur

Lenteur

Maîtrise ?





Rapidité

Exactitude

Stupidité

As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought.

Debugging had to be discovered.

I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

Maurice Wilkes, 1949

Le premier bug informatique...

9/9

0800 Arctan started
1000 " stopped - arctan ✓
1300 (032) MP - MC 1.2700 9.037 847 025
9.037 846 795 correct
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 110.00 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multy Adder Test.
1545 
Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1700 closed down.

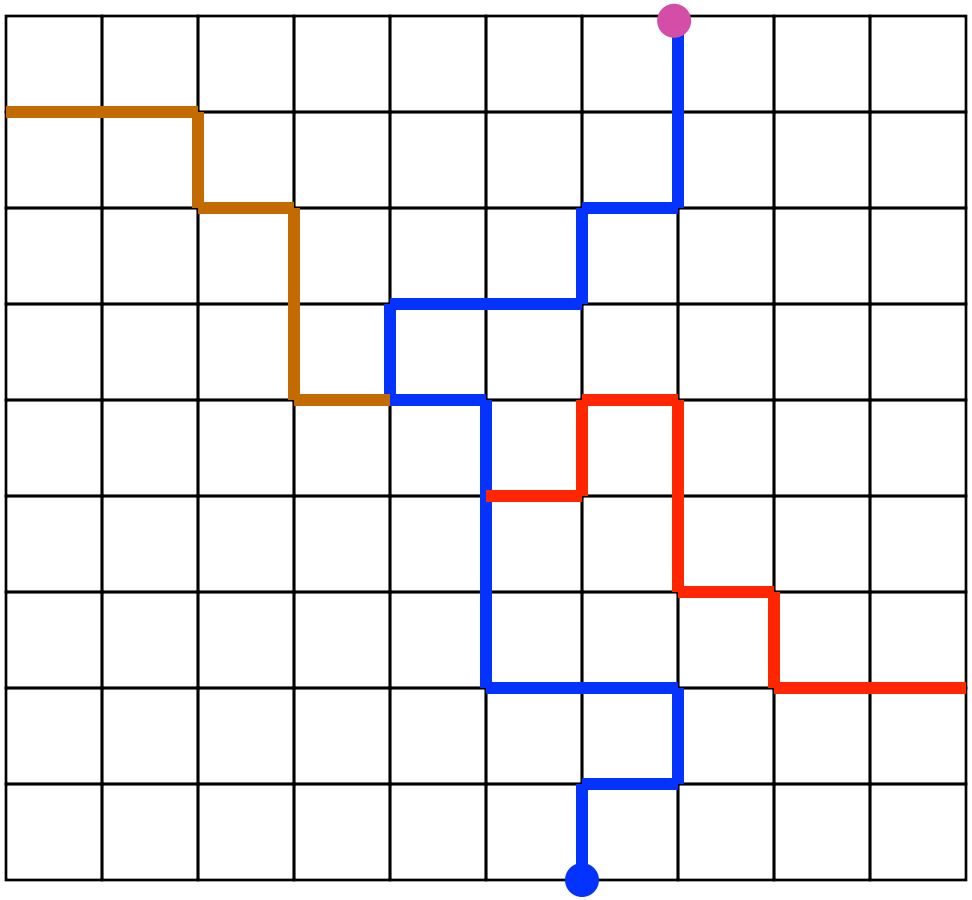
Grace Hopper, 9 septembre 1947, 15h45

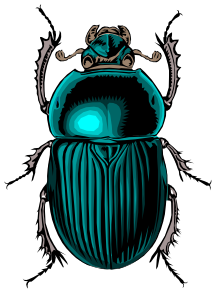
... mais le mot existait bien avant !

It has been just so in all of my inventions.
The first step is an intuition, and comes with a burst,
then difficulties arise — this thing gives out and [it is] then that
"Bugs" - as such little faults and difficulties are called -
show themselves and months of intense watching,
study and labor are requisite before commercial
success or failure is certainly reached.

Thomas Edison, 1878

Merci Wikipedia !

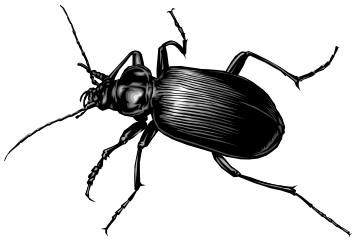




TDGGTDTTGDDTGDGT

TDGGTDTDGDDTGDGT

TDGGTDTTGTDTGDGT



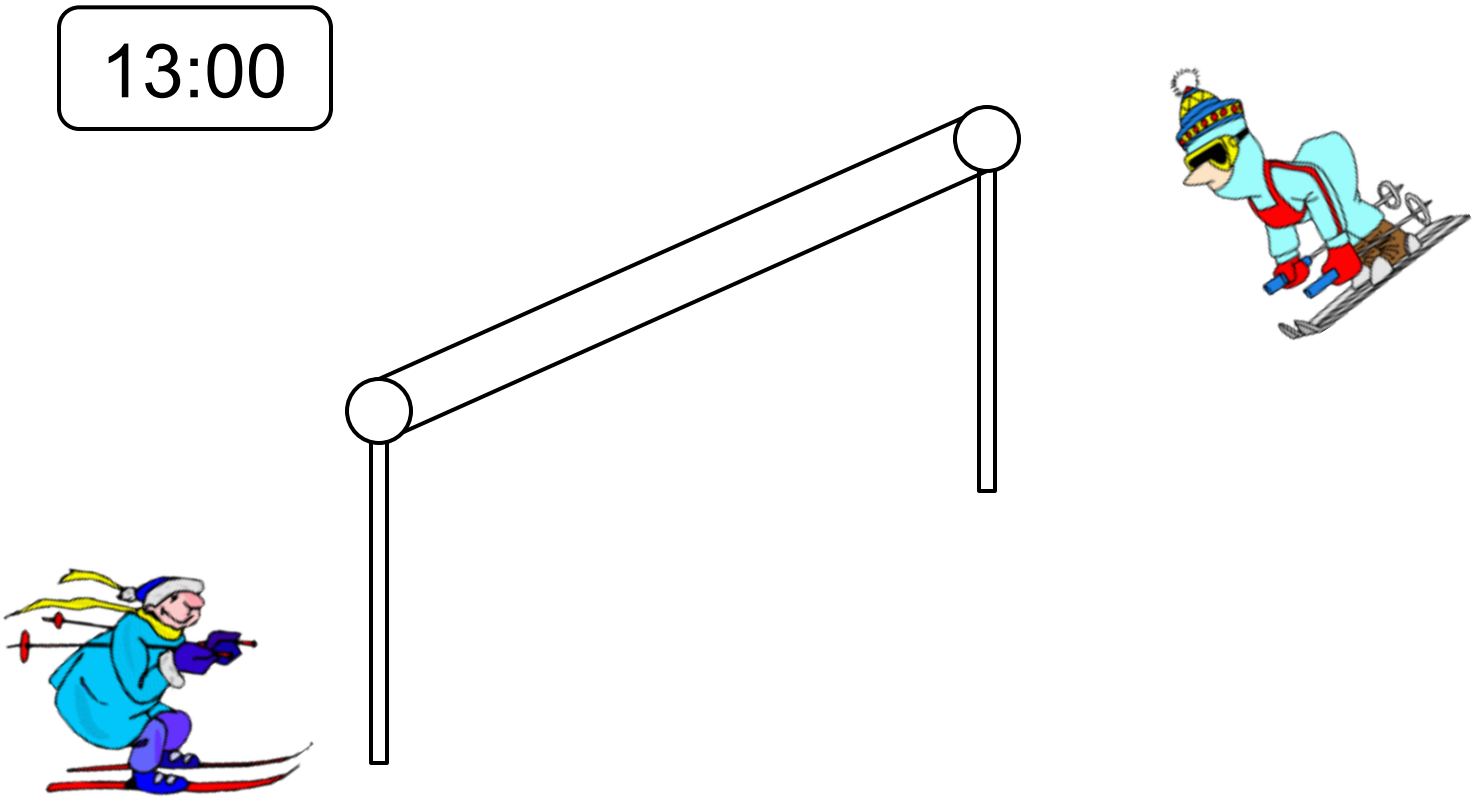
Interblocage (deadlock)

Lise et Laure

Famine (starvation)

Lise, Laure et Manon

Course entre processus: le ski au 20e siècle



cf.leçon inaugurale du 19/11/2009,
http://www.college-de-france.fr/site/gerard-berry/inaugural-lecture-2009-2010.htm

Comment nourrir les bugs

- Ne pas comprende les spécificités de l'informatique
  - ex. : logiciel vu comme « mécanique plus légère »
- Employer des raisonnements valables ailleurs mais pas en informatique
  - redondance par simple duplication du système
  - calculs de « probabilités de bugs »
- Mal spécifier, documenter, maintenir
  - mauvaises méthodes de travail, mauvais outils
- limiter les tests à la « marche normale »
  - les problèmes sont dans les coins et dans le non prévu
- Oublier de vérifier la vérification
  - en particulier qu'elle n'a pas de trous de couverture

Les problèmes des spécifications informelles

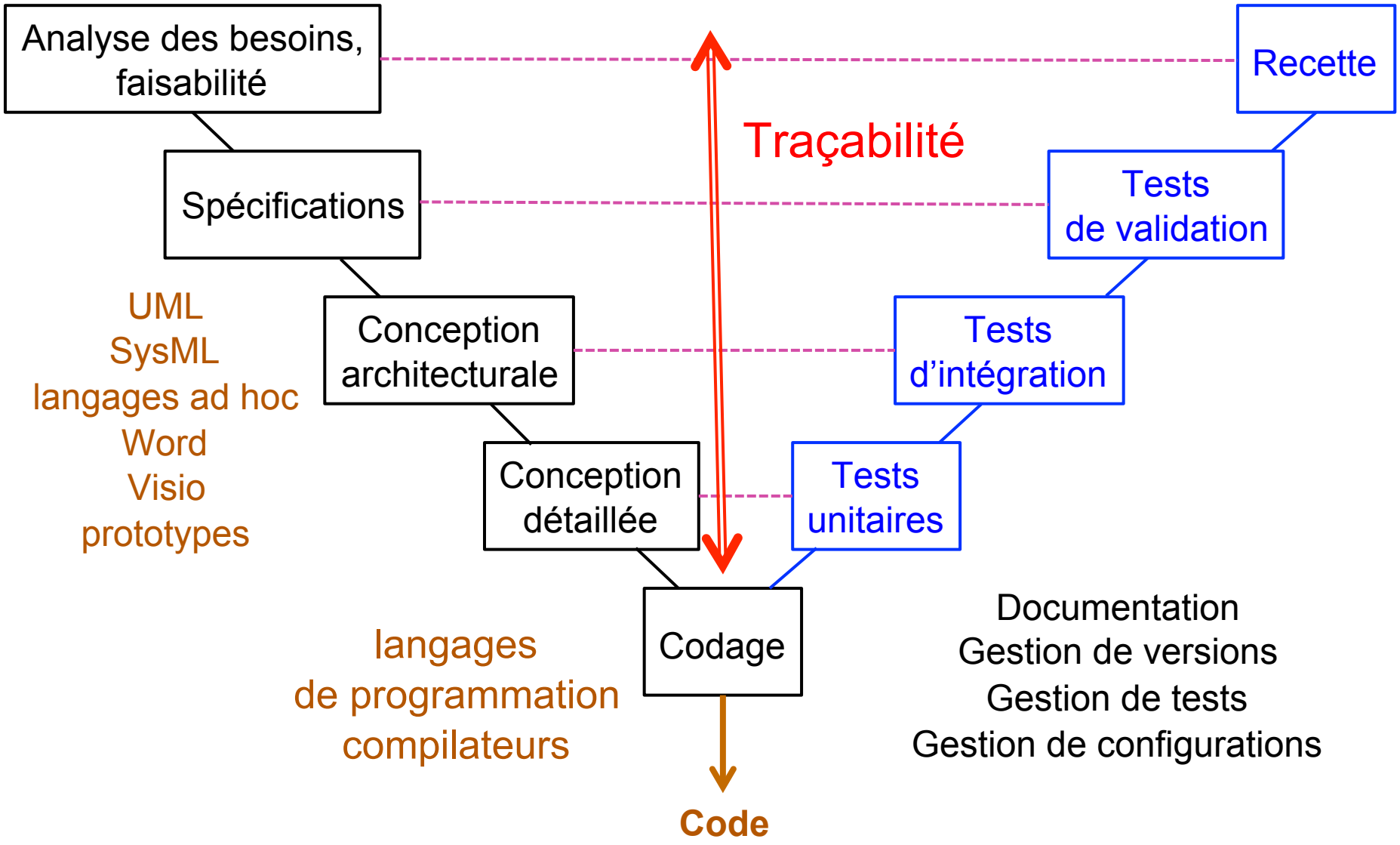
- Contradiction ou incohérence
  - page 12 : X est une variable entière positive
  - page 33 : si X est négatif, ...
- Sous-spécification
  - oublier de préciser les contraintes d'environnement du programme
  - oublier de spécifier précisément les cas d'erreurs
  - en cas de survenue d'alarmes multiples et rapprochées, le logiciel devra réagir en conséquence
- Sur-spécification
  - donner des détails inutiles compliquant la réalisation
    ex. : caractéristiques physiques irrelevantes,
    exigence d'un outillage de développement inadapté

Il est impossible de réaliser une bonne application
à partir de spécification incohérentes, ou même laides

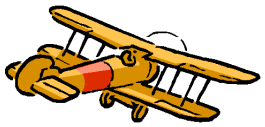
Agenda

1. Le bug, un danger de l'informatique
2. Comment réduire les bugs
3. Les méthodes de test : forces et faiblesses
4. La naissance de la vérification formelle
5. Les méthodes de vérification formelles
   1. Les assertions
   2. La réécriture symbolique
   3. La sémantique dénotationnelle
   4. Les logiques et assistants de preuve
   5. Le model-checking
6. Etat et perspectives

Le cycle en V traditionnel



Différents niveaux de vérification



pilotage, contrôle moteur, freinage carburant, etc.
vies humaines => critique certifié

trajectoire, attitude, imagerie, télécommunications
mission-critical => très haute qualité

téléphone, audio, TV, DVD, jeux
business critical => vitesse + qualité

pacemakers, contrôle d'insuline, robots chirurgiens
life-critical => vraie certification (j'ose espérer !)

De la conception à l'implémentation

analyse

spécification

langage ?
outillage ?

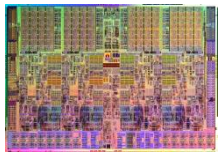
programmation

langage ?
outillage ?

intégration

outillage ?

implémentation



machine ?
OS ?

De la conception à l'implémentation

analyse

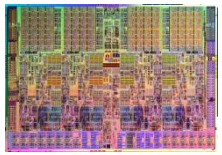
spécification

programmation

Nid à bugs

intégration

implémentation



Validation / Vérification

analyse

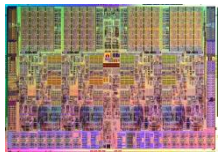
spécification

adéquation ?
consistance ?

programmation

intégration

implémentation



conformité ?
complétude ?

performances ?
sécurité ?

Comment réduire les bugs

- Utiliser de bonnes méthodes de design
  - spécifications, langages et déboguers à base formelle
  - caractérisant l'environnement autant que de l'application
- Rendre tout visible et explorable (= traçable)
  - procédures de développement, spécification, code, documentation, tests et résultats de test, etc.
- Faire des revues indépendantes
  - processus de certification, communautés open source, etc.
- Tester systématiquement
  - les composants, leur intégration, la non-régression
  - l'application globale sur la cible réelle ou par simulation

21e siècle : vérifier formellement
avec des outils automatiques ou semi-automatiques

Agenda

1. Le bug, un danger de l'informatique
2. Comment réduire les bugs
3. Les méthodes de test : forces et faiblesses
4. La naissance de la vérification formelle
5. Les méthodes de vérification formelles
   1. Les assertions
   2. La réécriture symbolique
   3. La sémantique dénotationnelle
   4. Les logiques et assistants de preuve
   5. Le model-checking
6. Etat et perspectives

Quelle « couverture » réalisent les tests ?

```
inst_0 ;
if test_1 then
   inst_1
end if ;
while exp do
   if test_2 then
      inst_2
   else
      if test_3 then
         inst_3
      end if;
      if test_4 then
          inst_4
      else
         inst_5
      end if
   end if
end while
```

couverture de spécifications

dans le cas A, faire B

couverture de code

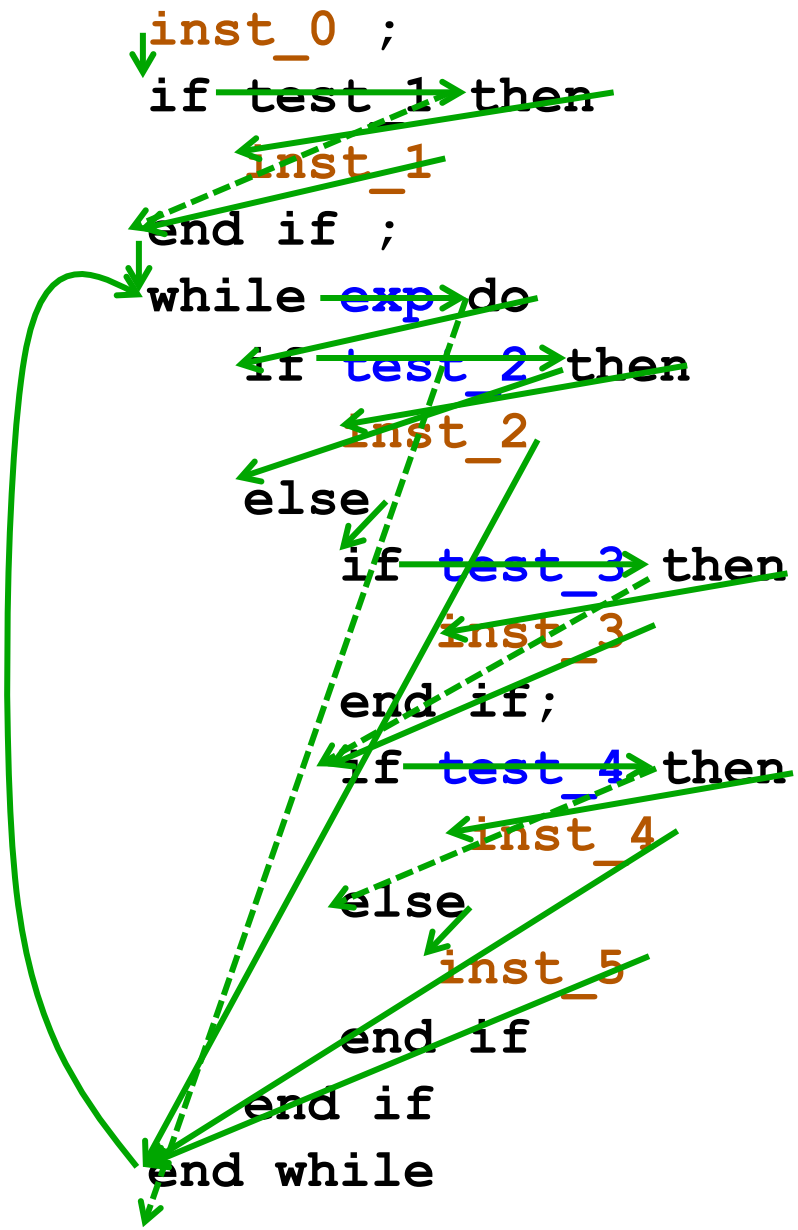
Couverture de code

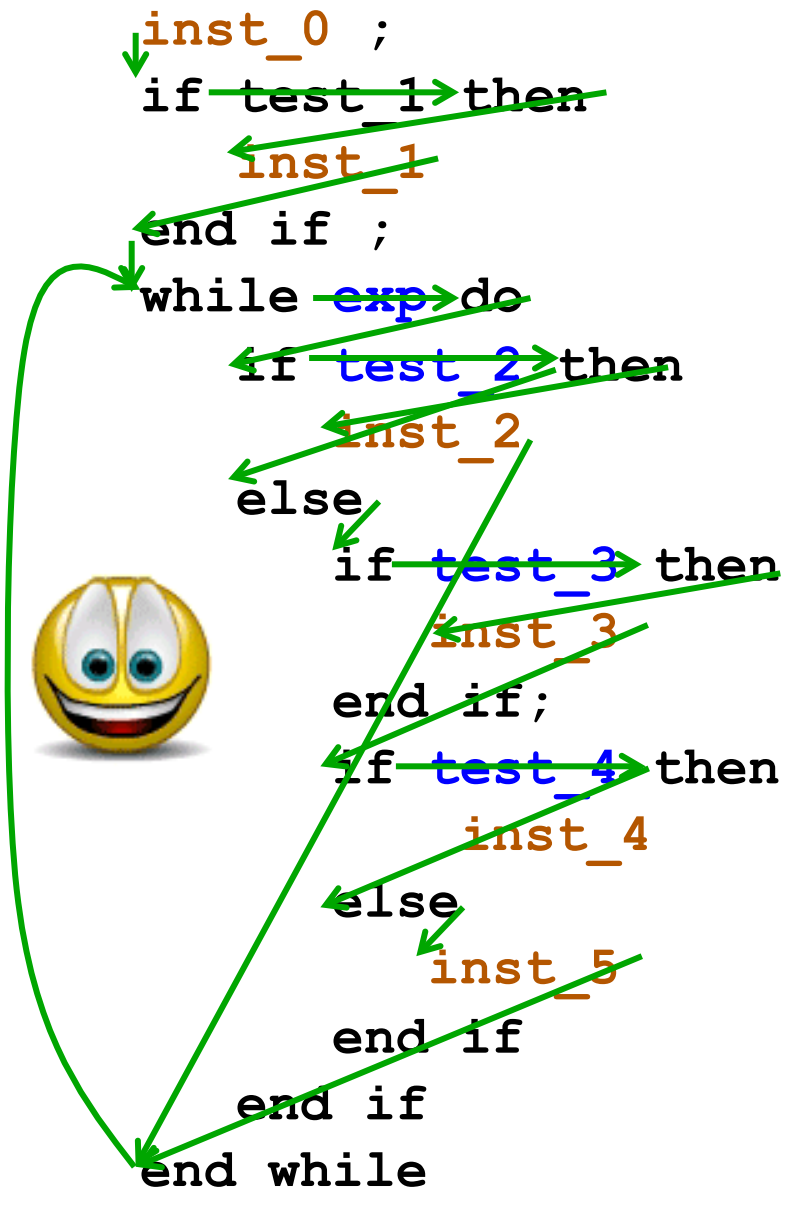
```
inst_0 ;
if test_1 then
   inst_1
end if ;
while exp do
   if test_2 then
      inst_2
   else
      if test_3 then
         inst_3
      end if ;
      if test_4 then
          inst_4
      else
         inst_5
      end if
   end if
end while
```

```
inst_0 ;
if test_1 then
   inst_1
end if ;
while exp do
   if test_2 then
      inst_2
   else
      if test_3 then
         inst_3
      end if ;
      if test_4 then
          inst_4
      else
         inst_5
      end if
   end if
end while
```

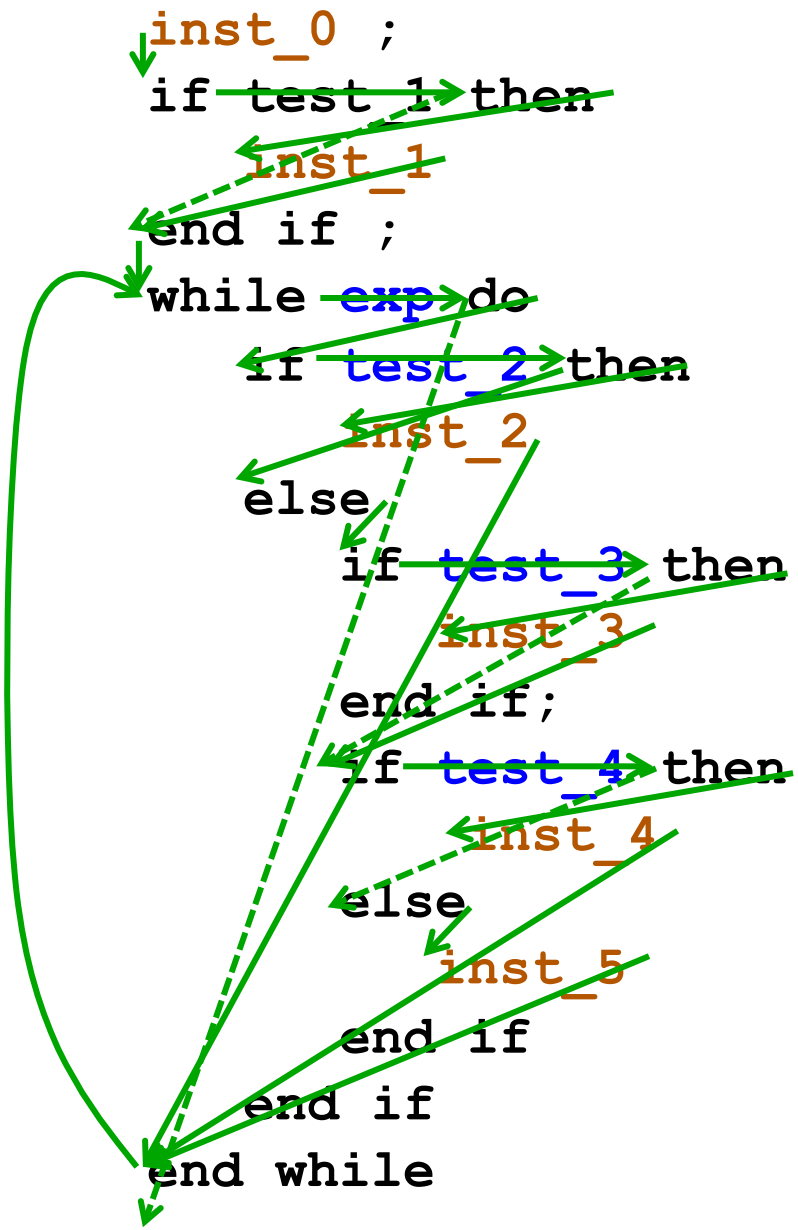
non testé ?
mort ?

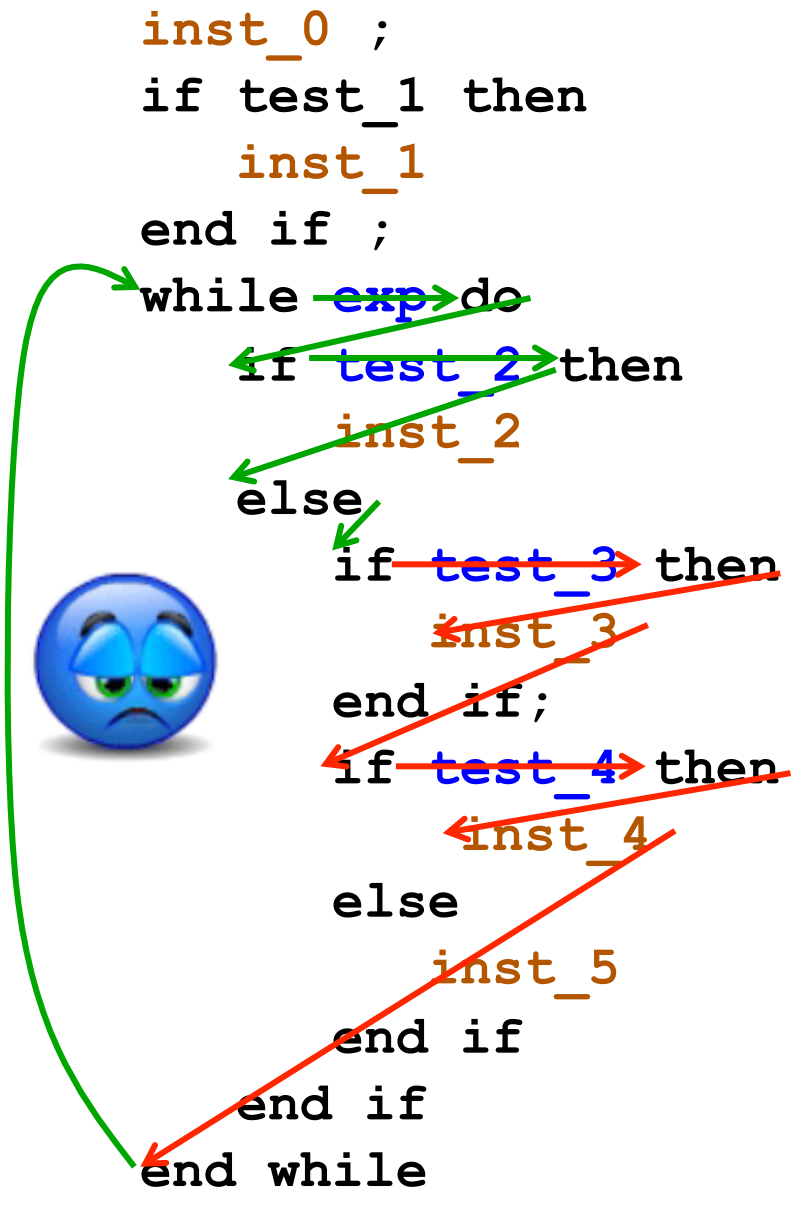
Couverture de chemins





Chemin inaccessible





Couverture de décisions

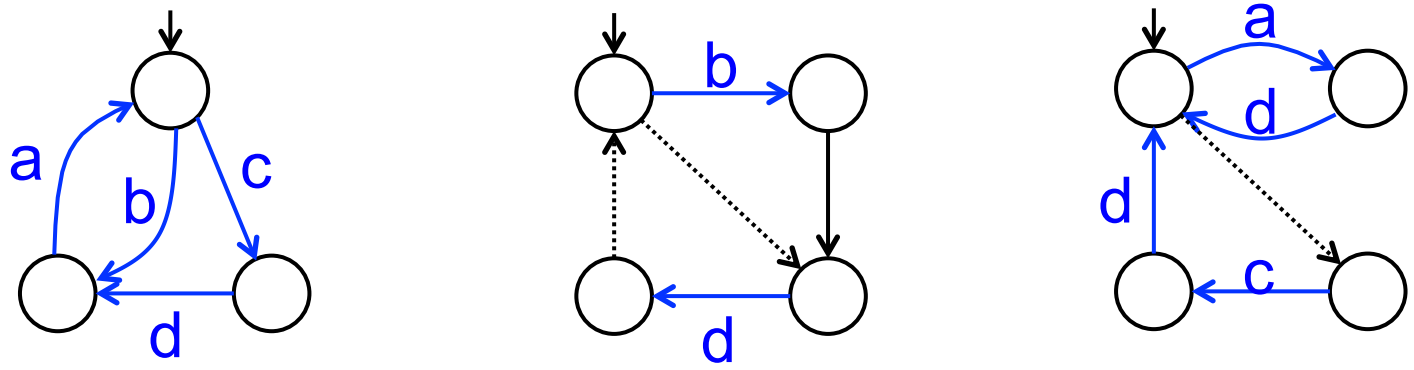
- if A or B then inst_1 else inst_2 end
  - A et B sont des conditions, « A ou B » une décision
  - 4 cas : ~~A^+B^+~~ , A^+B^-, A^-B^+, A^-B^-

⇒ réduction du nombre de tests sans perte de valeur

- Exemple : critère MCDC (Modified Condition / Decision Coverage), norme DO-178B appliquée à chaque décision
  - chaque condition a pris les valeurs vrai / faux au moins 1 fois
  - chaque décision a pris les valeurs vrai / faux au moins 1 fois
  - chaque condition a changé la valeur de la décision au moins une fois sans changer les valeurs des autres conditions

Intéressant, mais facile à pervertir en coupant les expressions à l'aide de variables intermédiaires...

Test de systèmes parallèles synchronisés (protocoles, circuits, algos distribués, etc.)



Les automates sont indépendants, sauf pour les actions a, b, c et d qui doivent être synchrones

- Les couverture indépendantes d'états et de transitions ne donnent que très peu d'informations

La vraie couverture des interactions peut être exponentielle, problème majeur !

Cf. cours des 18 et 25 mars 2015

Le test aléatoire

Produire des tests aléatoires complexes est un excellent moyen de traquer les bugs !

- Trois grandes méthodes
  - chercher à falsifier les assertions du programme
  - chercher à couvrir le code de différentes façons
  - utiliser le test différentiel, en comparant différents programmes devant normalement donner le même résultat
- Exemple d'outils
  - langage E, SystemVerilog pour les tests aléatoires dirigés de circuits
  - TGV de C. Jard *et. al*., DART de P. Godefroid
  - système Csmith pour la vérification de compilateurs C
  - et les model-checkers, voir cours 5 !

Test aléatoire de compilateurs C

Xuejun Yang, Yang Chen, Erich Eide et John Reger

Finding and Understanding Bugs in C Compilers

Proc. ACM SIGPLAN conf. on Programming Languages Design and Implementation (PLDI), San Jose (USA), June 2011

- C est devenu l'assembleur de haut niveau
  - beaucoup de programmes C sont engendrés automatiquement
  - les bugs de compilateur peuvent avoir des conséquences gravissimes
- Mais C est un langage complexe
  - expressions arithmético / logiques délicates, coercions et pointeurs arbitraires, arithmétique de pointeurs, bit-fields, etc.
  - 191 cas de comportements indéfinis, 52 cas de comportements non spécifiés, laissés au choix de l'implémentation
- Et les performances du code généré sont cruciales
  - beaucoup d'optimisations fondées sur des analyses statiques pas forcément justes...

Principes de Csmith

- Engendrer des programmes C complexes mais corrects
  - génération aléatoire à partir de la grammaire, filtrage par tests
  → aspects délicats : scoping, pointeurs, opérations, const, volatile, etc.
  - limitations : pas de chaînes, de flottants, d'unions, de récursion, d'allocation dynamique ni de pointeurs de fonctions

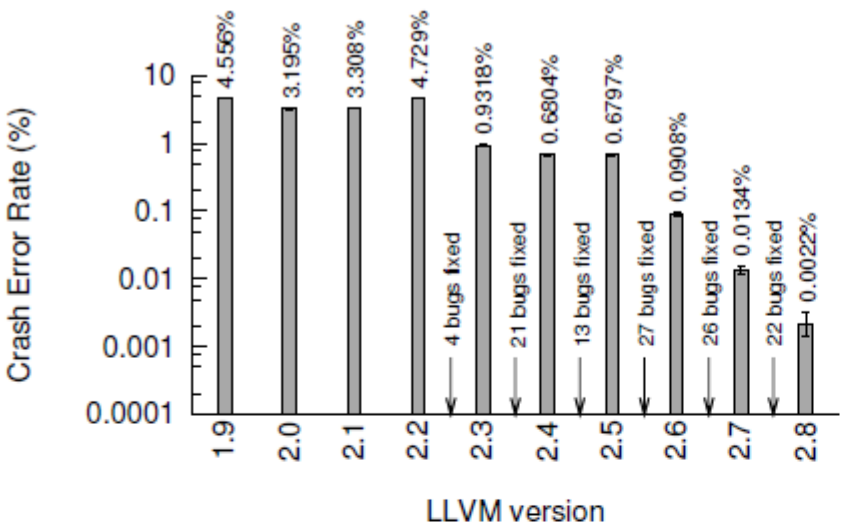
- Crasher et tester différentiellement les compilateurs
  - 1 million de programmes sources engendrés
  - détection directe des crashs et des erreurs internes
  - comparaison de l'exécution des codes générés par 12 compilateurs: GCC, LLVM, compilateurs commerciaux, CompCert (X. Leroy)

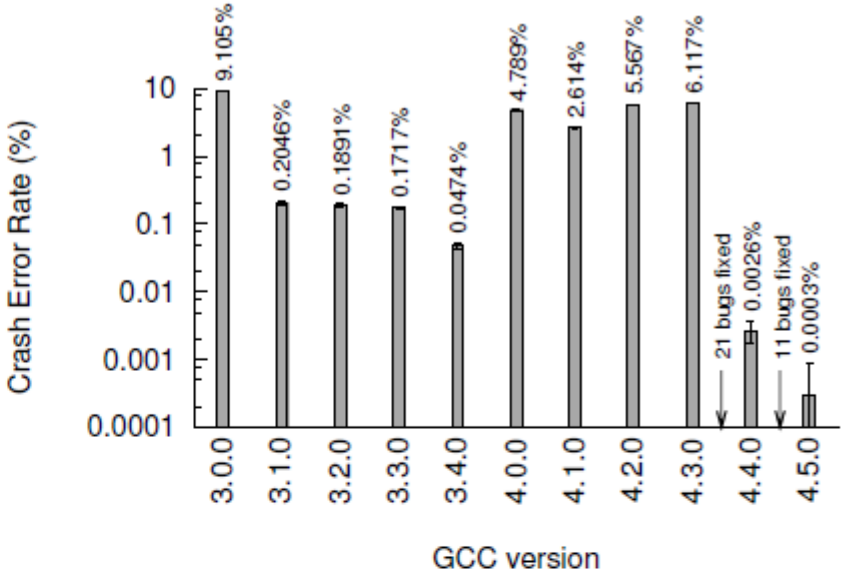
Plusieurs centaines de bugs détectés, surtout dans les phases intermédiaires (transformation et optimisation)

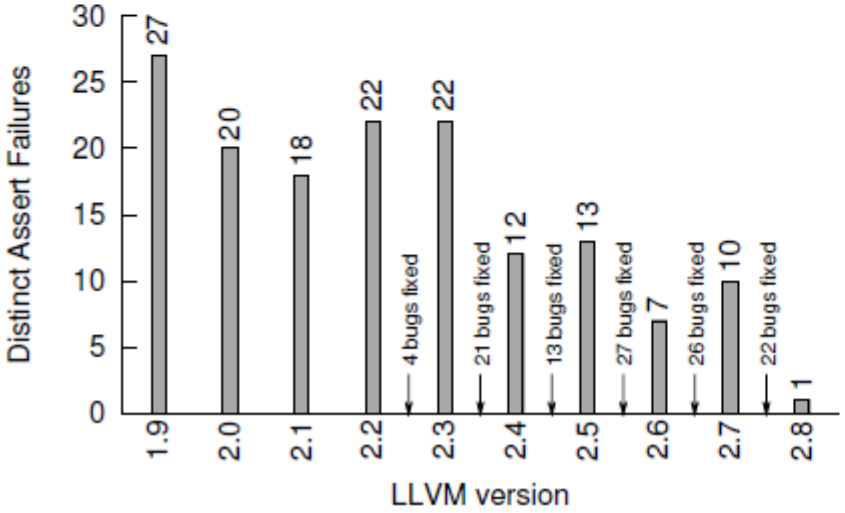
Des compilateurs différents ont des bugs différents !

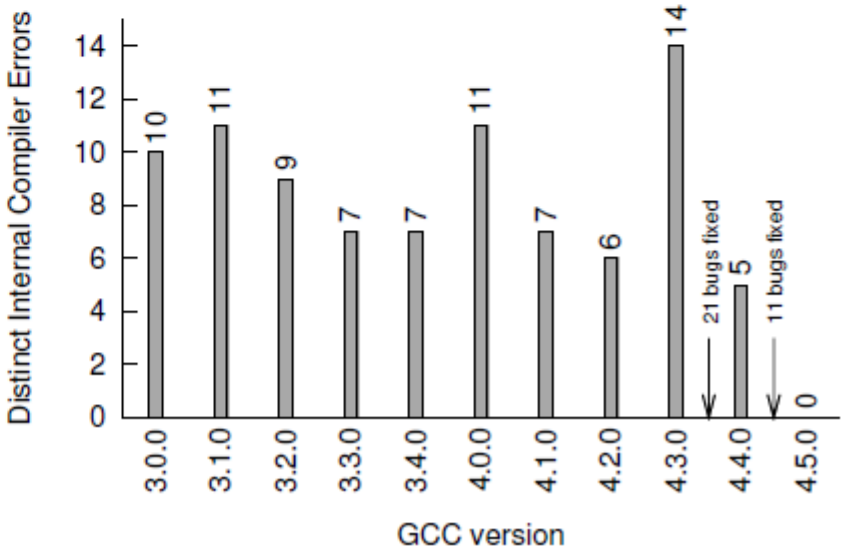
Un seul survivant : CompCert

Crashs et assertions internes violées



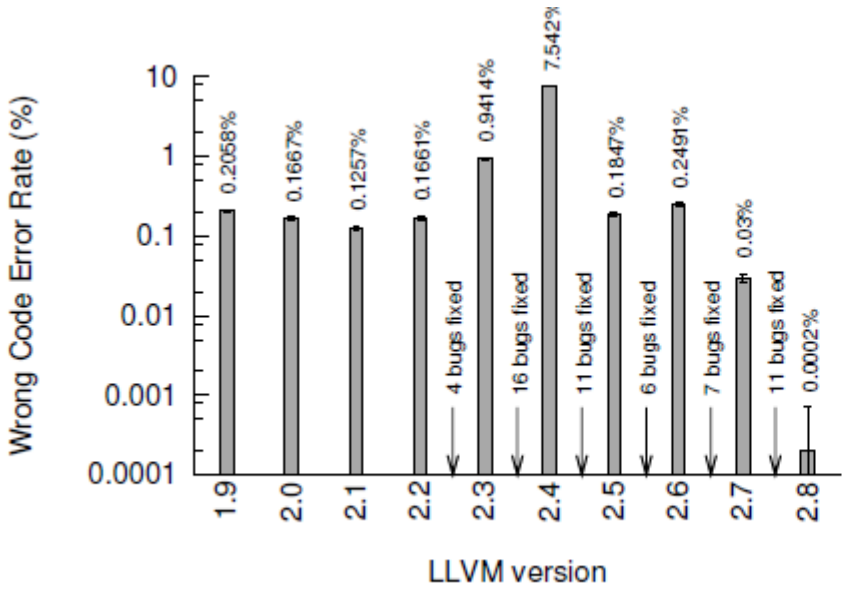


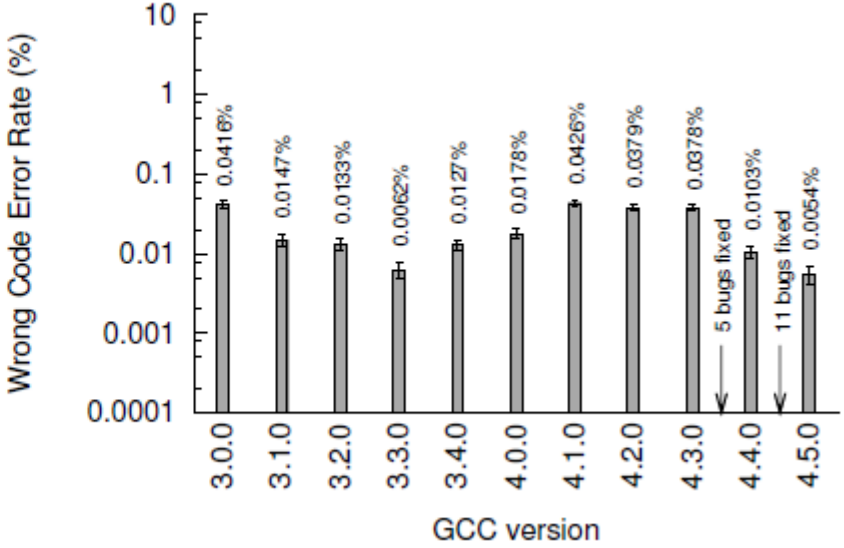




source Xuejun Yang, Yang Chen, Erich Eide et John Reger

Code généré faux





source Xuejun Yang, Yang Chen, Erich Eide et John Reger

CompCert de Xavier Leroy : 0 bug !

Etonnant? Non, il est certifié formellement (en Coq)

Agenda

1. Le bug, un danger de l'informatique
2. Comment réduire les bugs
3. Les méthodes de test : forces et faiblesses
4. La naissance de la vérification formelle
5. Les méthodes de vérification formelles
   1. Les assertions
   2. La réécriture symbolique
   3. La sémantique dénotationnelle
   4. Les logiques et assistants de preuve
   5. Le model-checking
6. Etat et perspectives

La remarque qui fonde la suite du cours

Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence

Edsger W. Dijkstra,
The Humble Programmer (1972)
Communications of the ACM ***15*** *(10), 972: pp. 859–866*

Peut-on faire mieux ? Oui !
En voyant la vérification de spécifications et de programmes comme des questions mathématiques à résoudre à l'aide de systèmes informatiques (eux-mêmes prouvés)

Attention : il faut des mathématiques vraiment formelles

Les devises de Leslie Lamport (1)*

- Writing is nature's way of letting you know how sloppy your thinking is

  Guindon

- Mathematics is nature's way of letting you know how sloppy your writing is

  Leslie Lamport, The TLA+ Book

- Formal mathematics is nature's way of letting you know how sloppy your mathematics is

  Leslie Lamport, The TLA+ Book

Alan Turing, 1949

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

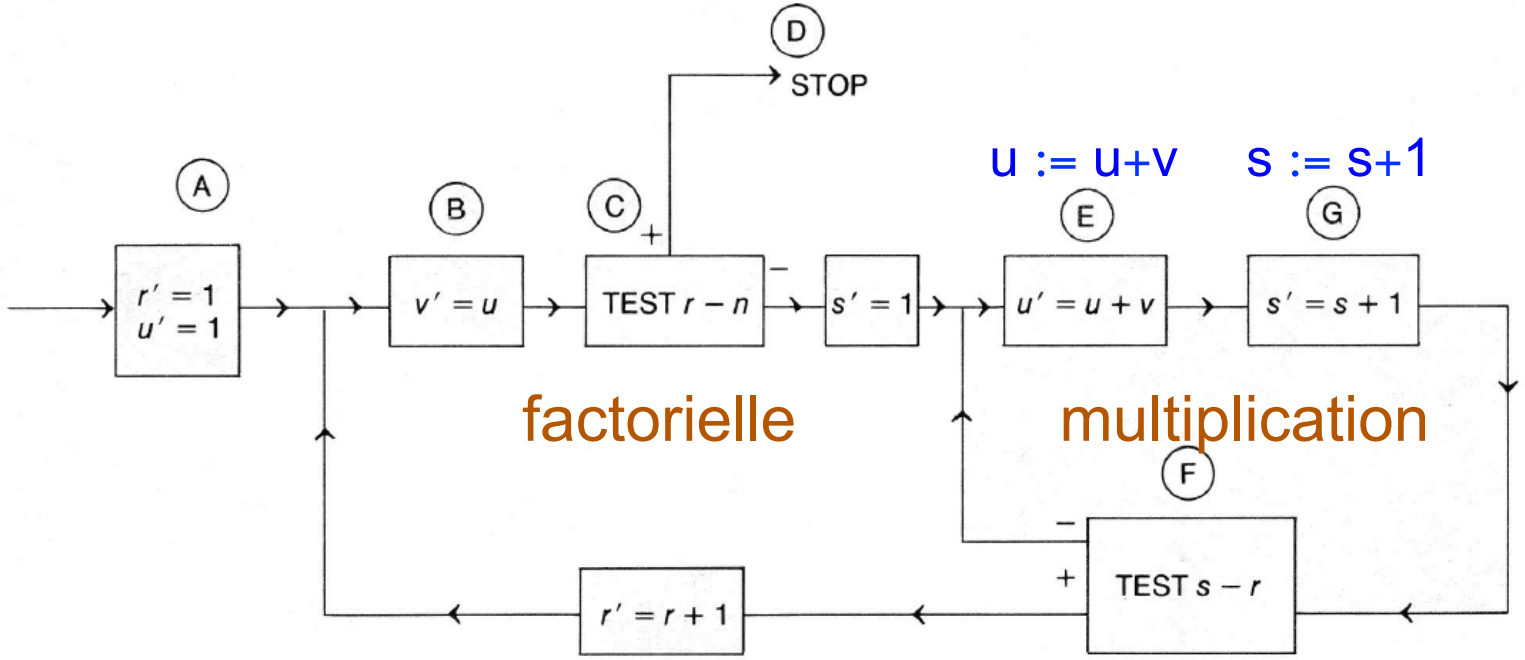


Table d'assertions

STORAGE LOCATION	(INITIAL) Ⓐ $k = 6$	Ⓑ $k = 5$	Ⓒ $k = 4$	(STOP) Ⓓ $k = 0$	Ⓔ $k = 3$	Ⓕ $k = 1$	Ⓖ $k = 2$
27					s	$s + 1$	s
28		r	r		r	r	r
29	n	n	n	n	n	n	n
30		$\lfloor r$	$\lfloor r$		$s \lfloor r$	$(s + 1) \lfloor r$	$(s + 1) \lfloor r$
31			$\lfloor r$	$\lfloor n$	$\lfloor r$	$\lfloor r$	$\lfloor r$
	TO Ⓑ WITH $r' = 1$ $u' = 1$	TO Ⓒ	TO Ⓓ IF $r = n$ TO Ⓔ IF $r < n$	↑ n!	TO Ⓖ	TO Ⓑ WITH $r' = r + 1$ IF $s \geq r$ TO Ⓔ WITH $s' = s + 1$ IF $s < r$	TO Ⓕ

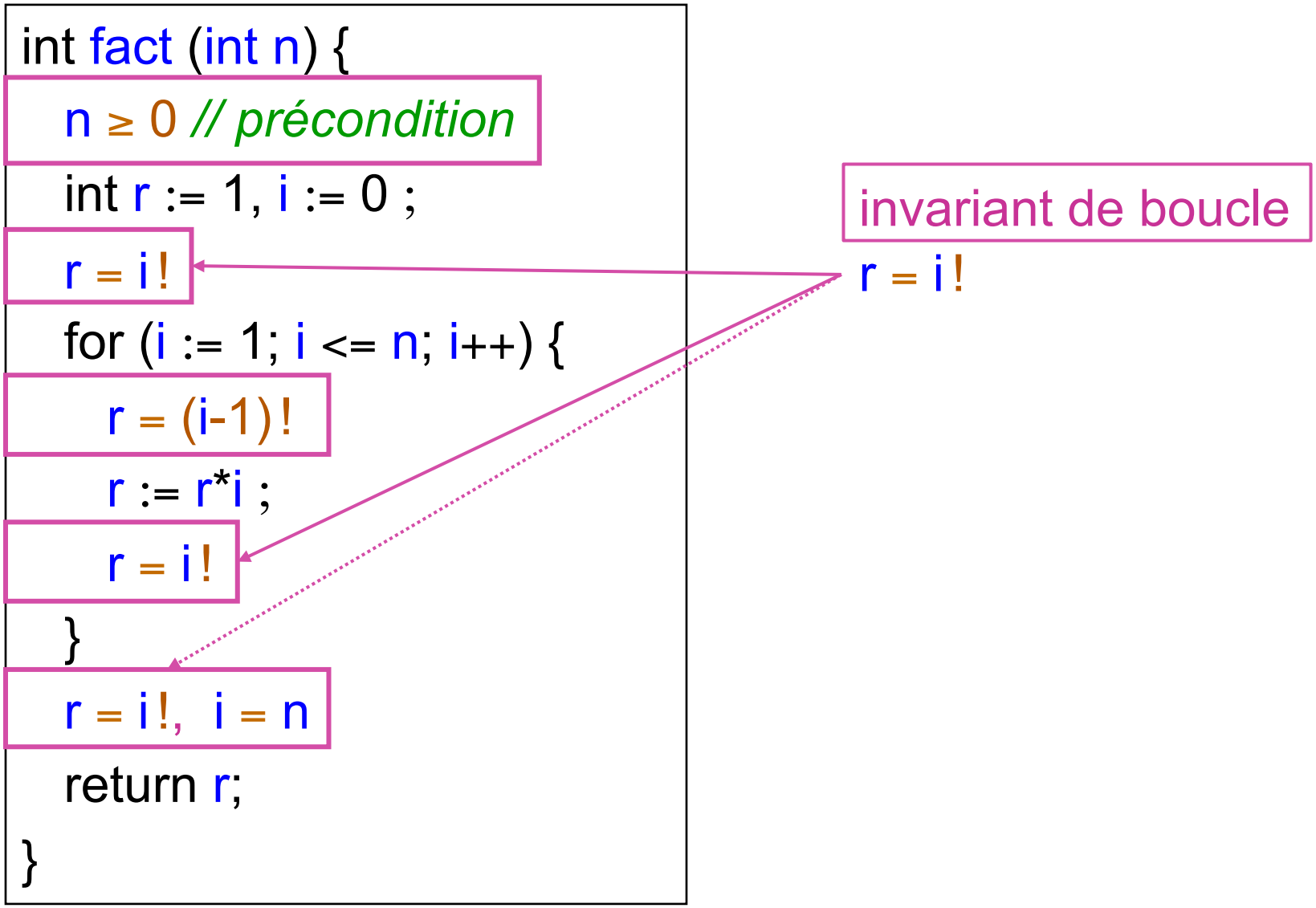
Figure 2 (Redrawn from Turing's original)

terminaison : décroissance d'un ordinal, voir cours du 11/03/2015

Agenda

1. Le bug, un danger de l'informatique
2. Comment réduire les bugs
3. Les méthodes de test : forces et faiblesses
4. La naissance de la vérification formelle
5. Les méthodes de vérification formelles
   1. Les assertions
   2. La réécriture symbolique
   3. La sémantique dénotationnelle
   4. Les logiques et assistants de preuve
   5. Le model-checking
6. Etat et perspectives

Encore la factorielle (cours du 22/02/2008)



Les assertions, un outil devenu fondamental

- Turing* → Floyd* / Hoare* → Dijkstra* → tout le monde...
  la difficulté est bien sûr dans les boucles et la récursion
- Base directe des langages à contrats et de leurs vérifieurs :
  Eiffel (B. Meyer), Spec# (M. Barnett, R. Leino & W. Schulte),
  Key-JML (Chalmers, Darmstadt, cf. TimSort bug), etc.
- Utiles en débogage classique, car vérifiables à l'exécution

Il faut s'arrêter dès qu'un problème est détecté,
sinon le circuit exécute toute la suite du code,
qui peut être arbitrairement dangereuse,
avec une conscience professionnelle absolue !

* : Turing Award

Agenda

1. Le bug, un danger de l'informatique
2. Comment réduire les bugs
3. Les méthodes de test : forces et faiblesses
4. La naissance de la vérification formelle
5. Les méthodes de vérification formelles
   1. Les assertions
   2. La réécriture symbolique
   3. La sémantique dénotationnelle
   4. Les logiques et assistants de preuve
   5. Le model-checking
6. Etat et perspectives

McCarthy : fonctions et réécriture*

A Basis For a Mathematical Theory of Computation

Computer Programming and Formal Systems, North-Holland (1963)

Computation is sure to become one of the most important of the sciences. This is because it is the science of how machines can be made to carry out intellectual processes. We know that any intellectual process that can be carried out mechanically can be performed by a general purpose computer.

Moreover, the limitations on what we have been able to make computers do so far come far more from our weakness as programmers than from the intrinsic limitations of the machines. We hope that these limitations can be greatly reduced by developing a mathematical theory of computation.

*Le programme de McCarthy**

1. Un langage de programmation universel
   ALGOL ? Lisp ? Pascal ?
   Mais finalement, tous pareils → pseudo-code
2. Une théorie de l'équivalence de programmes
   ⇒ transformations de programmes garanties sûres
   ⇒ aller sûrement de l'écriture initiale à une version efficace
3. Une représentation symbolique souple des algorithmes
   syntaxe abstraite
4. Une formalisation des ordinateurs et de leurs calculs
   cf. cours 2009-2010
5. Un trajet vers une théorie quantitative du calcul
   → la complexité algorithmique

La science en 1963

Théorie de la calculabilité

Trop négative

modèles trop abstraits

Théorie des automates

Trop limitée

Théorie des langages de programmation

En enfance (ALGOL)

Analyse numérique

Trop spécialisée

Equations et réécriture automatique de formules

De Kleene à Recursion Induction

Définition : $m + n = (n=0 \rightarrow m \ , \ T \rightarrow (m' + n^-))$

$m \ \text{W} \ n = (n=0 \rightarrow 0 \ , \ T \rightarrow (m + (m \ \text{W} \ n^-)))$

Théorème 1 : $(m + n)' = m' + n$

Preuve : soit $f(m,n) = (n=0 \rightarrow m' \ , \ T \rightarrow f(m',n^-)')$

1. par récurrence sur n, f termine toujours pour $m,n \in N$
2. par réécriture des termes, les deux fonctions $g(m,n) = (m + n)'$ et $h(m,n) = m' + n$ satisfont l'équation de f. Donc $g = h$. □

Théorème 2 : $m + 0 = m$

Preuve : demande 3 axiomes supplémentaires :

$m' \neq 0$, $(m')^- = m$, $(m \neq 0) \supset (m^-)' = m$

McCarthy 1963 : le plein de nouvelles notions !

- Programmation fonctionnelle récursive : bien plus simple !
- Syntaxe abstraite : programmes → structures de données
- Preuves par réécritures de termes : mécanisables
- Importance du choix des axiomes : Peano, ZF, etc.
- Difficulté intrinsèque de la terminaison : indécidable

Successeurs
Boyer&Moore, ACL2, MAUDE, REVE, ProVerif, etc.

Agenda

1. Le bug, un danger de l'informatique
2. Comment réduire les bugs
3. Les méthodes de test : forces et faiblesses
4. La naissance de la vérification formelle
5. Les méthodes de vérification formelles
   1. Les assertions
   2. La réécriture symbolique
   3. La sémantique dénotationnelle
   4. Les logiques et assistants de preuve
   5. Le model-checking
6. Etat et perspectives

La sémantique dénotationnelle

- P. Landin, 1966 : ISWIM = If You See What I Mean
  λ-calcul + fondations sémantiques

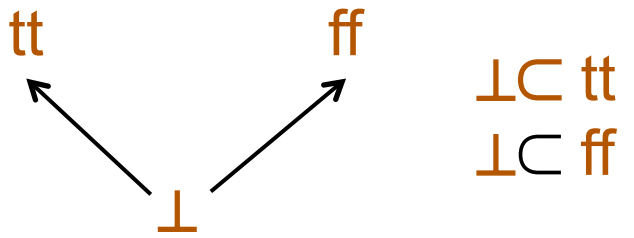
- D. Scott*, 1970 : D^{∞}, le premier modèle du λ-calcul pur
- D. Scott*, C. Strachey : sémantique dénotationnelle par le λ-calcul et la logique de Scott
- G. Plotkin, 1970, R. Milner*, 1974 : PCF et ses modèles

→ G. Berry, P-L. Curien, A. Stoughton, S. Abramsky : théorie des langages séquentiel

Cf. cours des 2 et 9 décembre 2009

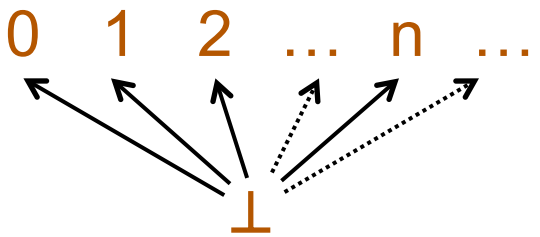
*Les domaines de Scott**

Ordre d'information : $x \subset y \leftrightarrow x$ a moins d'information que y



$O_\perp$: espace de Sierpinski

$B_\perp$: booléens de Scott



$\mathbf{N}_\perp$: entiers de Scott

Fonctions totales et croissantes

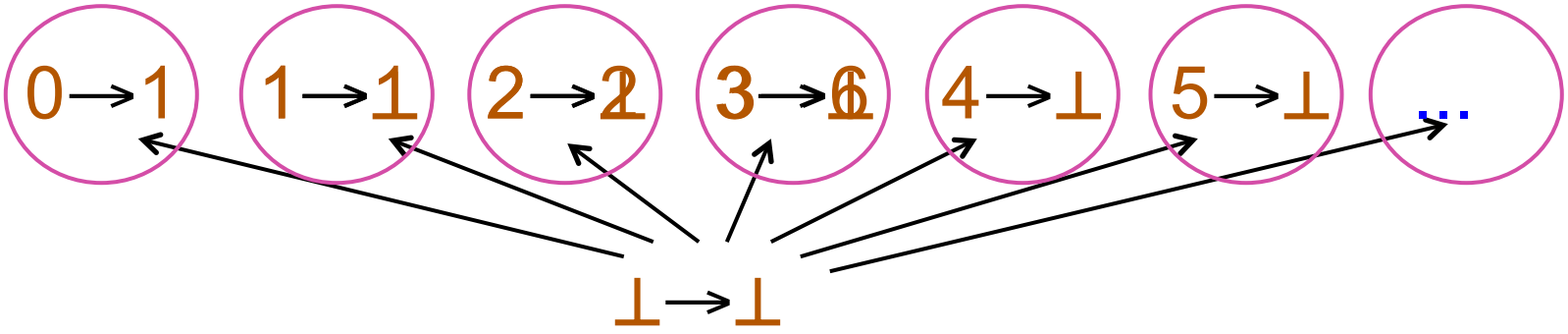
- Plus on en donne, plus on en récupère !

  f : <D,⊂,⊥> → <D',⊂',⊥'> croissante

  ssi ∀x, y. x ⊂ y ⇒ f(x) ⊂' f(y)

- Calcul itératif jusqu'à point fixe

  fact(n) = si n=0 alors 1 sinon n * fact(n-1)



Deux conséquences majeures

- R. Milner*, M. Newey, Weirauch : premier assistant de preuve

  1970 : LCF = Logic For Computable Functions

  Stanford LCF → Edinburgh LCF (G. Huet, L.Paulson)

  → langages typés ML, Caml, Haskell, etc.

  mais manipuler les ⊥ est trop compliqué...

- P. Cousot et R. Cousot, 1977 : interprétation abstraite

  Théorie générale d'approximations de données

  algorithmes efficaces de calcul des approximations

  ex: x/y ok si y ∈ [u,v] avec 0 ∉ [u,v]

  x[i] hors des bornes, etc.

  devenue industriel, applications majeures

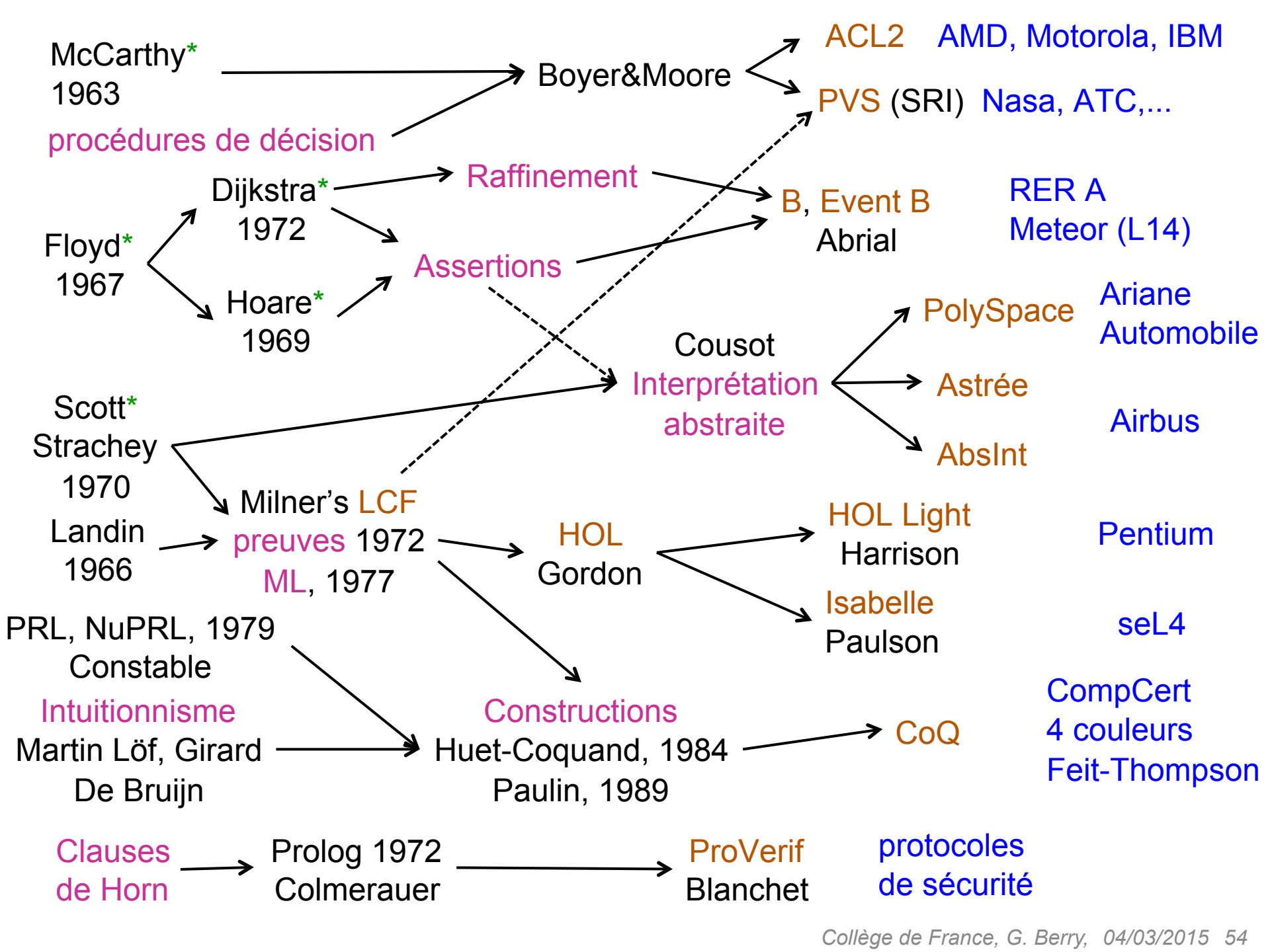
Agenda

1. Le bug, un danger de l'informatique
2. Comment réduire les bugs
3. Les méthodes de test : forces et faiblesses
4. La naissance de la vérification formelle
5. Les méthodes de vérification formelles
   1. Les assertions
   2. La réécriture symbolique
   3. La sémantique dénotationnelle
   4. Les logiques et assistants de preuve
   5. Le model-checking
6. Etat et perspectives

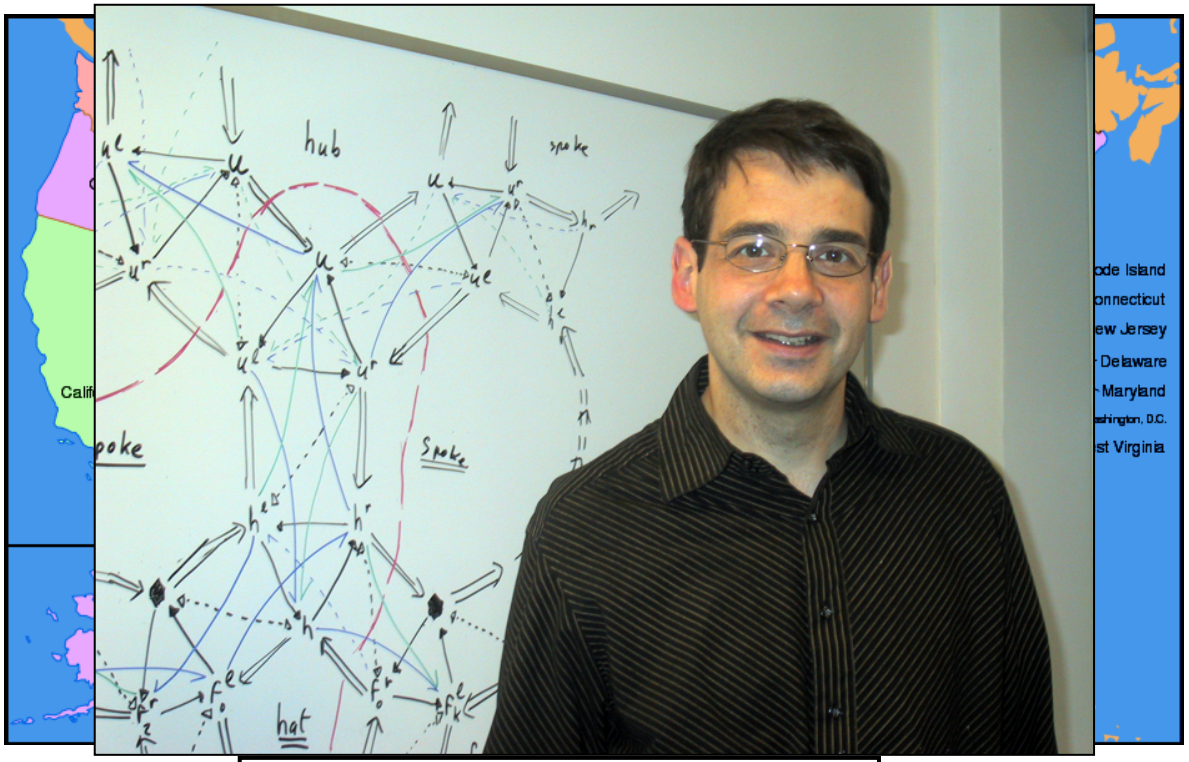
Vérification par assistants de preuves

- Logiques formelles :
  - calcul des prédicats en B, Event-B
  - calcul des prédicats d'ordre supérieur en HOL
  - calcul des constructions inductives en Coq
  - métalogiques en Isabelle
- Des algorithmes fondamentaux : typage, unification, etc.
- Gestion de grandes preuves
- Applications: HW et SW complexe, théorèmes mathématiques

Sont en train de passer à l'échelle des vrais problèmes,
en informatique et en mathématiques


McCarthy*
1963
procédures de décision
Boyer&Moore
ACL2
AMD, Motorola, IBM
PVS (SRI)
Nasa, ATC,...
Dijkstra*
1972
Raffinement
B, Event B
Abrial
RER A
Meteor (L14)
Floyd*
1967
Assertions
Hoare*
1969
Cousot
Interprétation
abstraite
PolySpace
Ariane
Automobile
Astrée
Airbus
AbsInt
Scott*
Strachey
1970
Landin
1966
Milner's LCF
preuves 1972
ML, 1977
HOL
Gordon
HOL Light
Harrison
Pentium
Isabelle
Paulson
seL4
PRL, NuPRL, 1979
Constable
Intuitionnisme
Martin Löf, Girard
De Bruijn
Constructions
Huet-Coquand, 1984
Paulin, 1989
CoQ
CompCert
4 couleurs
Feit-Thompson
Clauses
de Horn
Prolog 1972
Colmerauer
ProVerif
Blanchet
protocoles
de sécurité

Les mathématiques sur ordinateur



- 1852 Guthrie
- 1976 Appel – Haken
- 2005 Gonthier (en Coq)

Preuve omise, évidente mais longue

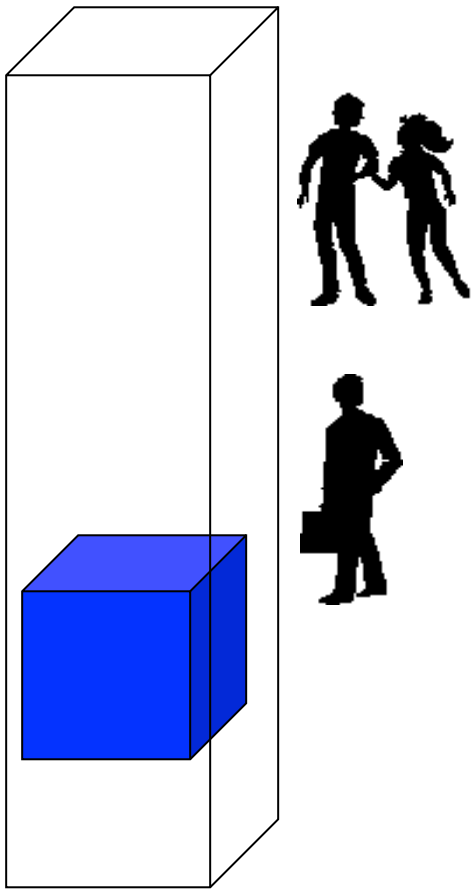
2013 : Feit-Thompson's Odd Order Theorem

255 pages de maths lourdes → Coq !

Agenda

1. Le bug, un danger de l'informatique
2. Comment réduire les bugs
3. Les méthodes de test : forces et faiblesses
4. La naissance de la vérification formelle
5. Les méthodes de vérification formelles
   1. Les assertions
   2. La réécriture symbolique
   3. La sémantique dénotationnelle
   4. Les logiques et assistants de preuve
   5. Le model-checking
6. Etat et perspectives

Model-checking temporel



Spécification : doit laisser beaucoup de liberté, plusieurs algorithmes possibles

Sûreté : l'ascenseur ne voyage jamais la porte ouverte

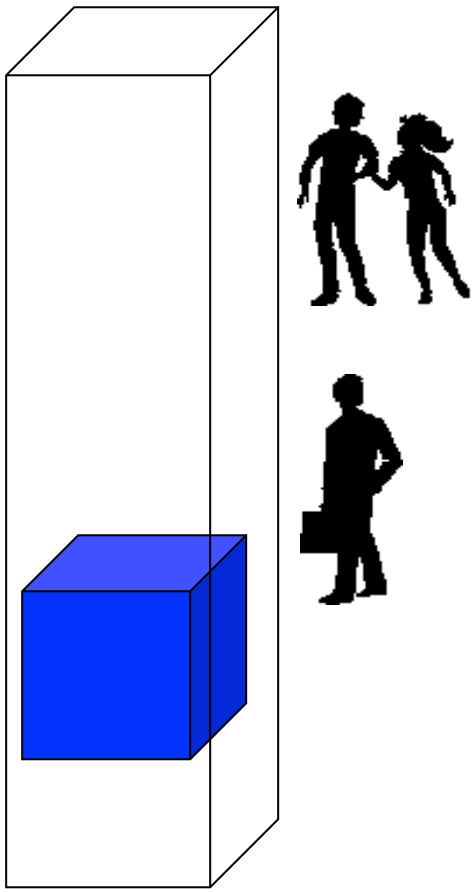
$$\square\,(\text{Bouge} \Rightarrow \text{PorteFermée})$$

$\square$: toujours

Bouge, PorteFermée : prédicats instantanés

Génération de contre-exemples pour les propriétés fausses

Model-checking temporel



Spécification : doit laisser beaucoup de liberté, plusieurs algorithmes possibles

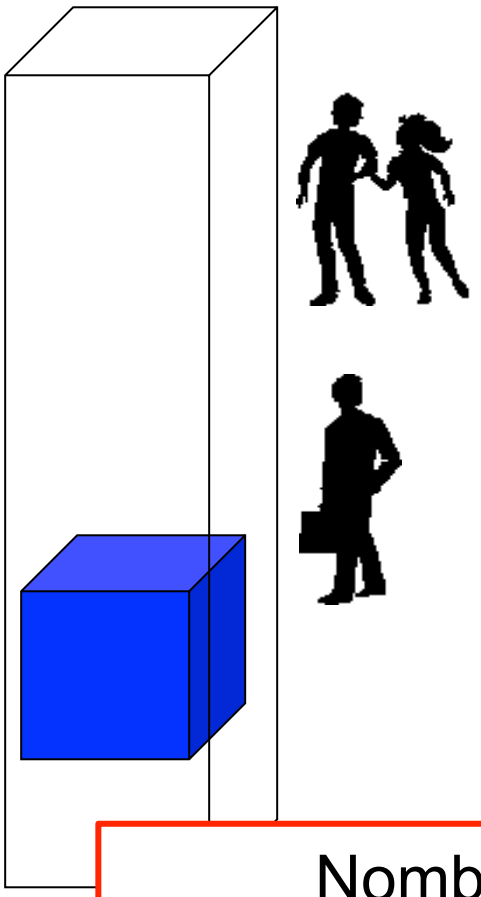
Vivacité : l'ascenseur finit par s'arrêter à tout étage où il a été appelé

$$\Box(\text{Appel} \Rightarrow \Diamond \text{Arrivée})$$

toujours — un jour

Plus dur, car parle du futur à distance quelconque
testable si nb d'étages connus,
pas si c'est un paramètre

Model-checking temporel



Spécification : doit laisser beaucoup de liberté, plusieurs algorithmes possibles

Génération automatique de tests

Question négative : il est impossible de visiter tous les étages

Réponse négative : faux! Il suffit de faire la séquence suivante : .....

Résultat positif : Merci !

Nombreuses applications industrielles :
CAO de circuits, protocoles, algorithmes distribués, réseaux sur puce, logiciels critiques, temps-réel, etc.

Les Model Checkers

- Méthodes énumératives :
  - A. Pnueli* : logiques temporelles
  - JP. Queille, J. Sifakis* : CESAR
  - E. Clarke*, E. Emerson* : EMC

  | naissance du model-checking

  - G. Holzmann : SPIN
  - H. Garavel, R. Mateescu, etc : CADP
  - D. Dill : MurΦ

Les Model Checkers

- Méthodes de calcul symbolique

  R. Bryant : BDDs → K. McMillan : EMC → SMV
  → B. Kurshan : COSPAN
  → JC. Madre, O. Coudert, H. Touati : TiGeR
  → Esterel v5 / v7, cf cours du 01/04

  SAT / SMT → Z3, NuSMV, ABC, Alt-Ergo, etc.

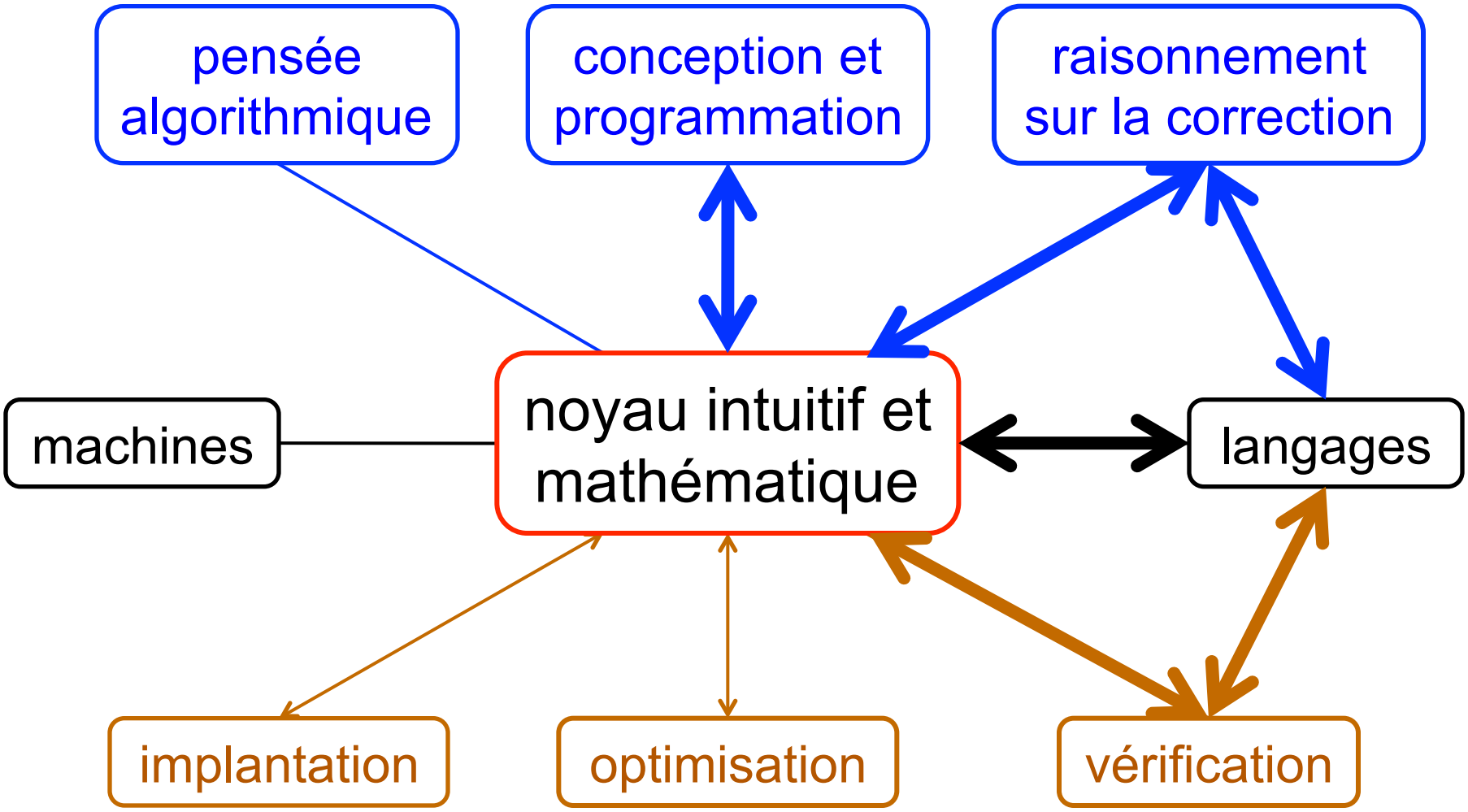
- Vérification de systèmes temps-réel :

  R. Kurshan, R. Alur : COSPAN
  T. Henzinger : HyTech
  K. Larsen : UPPAAL

Agenda

1. Le bug, un danger de l'informatique
2. Comment réduire les bugs
3. Les méthodes de test : forces et faiblesses
4. La naissance de la vérification formelle
5. Les méthodes de vérification formelles
   1. Les assertions
   2. La réécriture symbolique
   3. La sémantique dénotationnelle
   4. Les logiques et assistants de preuve
   5. Le model-checking
6. Etat et perspectives

Anatomie d'une approche formelle moderne



Objectifs

- Remplacer ou compléter le test par des preuves mathématiques, avec au moins le degré de confiance associé aux maths.

- Automatiser les preuves autant que possible
- Produire des contre-exemples pour les propriétés fausses
- Rendre la preuve accessible à des ingénieurs normaux
- Intégrer la preuve dans les flots de développement usuels
- Comprendre quand l'effort de preuve est justifié en pratique

Etat de l'art

- Test et preuve se complètent
  - l'équilibre dépend de la criticité
- Le model-checking est devenu d'usage courant
  - circuits, systèmes distribués, etc.
- Les assistants de preuve montent en puissance
  - Métros, Pentium, seL4, CompCert, algorithmes distribués, sécurité, etc.
  - maths: 4 couleurs, Feit-Thompson (Gonthier *et. al.*), Kepler (Hales), etc.
- Mais il reste beaucoup à faire pour un usage plus général

Quoi vérifier ?

- Conformité de l'action au besoin?

- Vérification complète
  - cohérence interne des spécifications
  - conformité de la réalisation aux spécifications
- Vérification d'équivalence ou raffinement
  - passage d'un code qui marche à un code bien écrit
  - vérification d'optimisations

- Vérification partielle de points clefs
  - terminaison
  - absence d'erreur à l'exécution
  - vérification d'assertions locales ou gobales (syst. distribués)

Quand utiliser la vérification formelle

Le plus tôt possible !

- Si possible, dès la spécification
  - description formelle (et modulaire) de la fonction, de l'environnement et des contraintes à respecter
- Et dans tout le passage de la spécification à la réalisation
  - raffinement graduel ou preuves de conformité
- Sinon (contraintes industrielles), dès le début de la réalisation
  - remise en forme des spécification, en particulier sur les points douteux
  - validation formelle dès le début du codage (ex. Esterel)
- En mode pompier, c'est beaucoup plus difficile....
  - mais fort utile pour la promotion du sujet
  - crash téléphone US, Ariane 5, bug Pentium, bugs de drivers, pbs sécurité, etc.

Bibliographie historique (1)

F.L. Morris and C. B. Jones
An Early Program Proof by Alan Turing
Annals of the History of Computing, Volume 6, Number 2, April 1984

John McCarthy
A Basis For a Mathematical Theory of Computation
Computer Programming and Formal Systems, North-Holland (1963)

Robert Floyd
Assigning Meanings to Programs
Proceedings of Symposium on Applied Mathematics, Vol. 19 (1967)

C. A. R. Hoare
An Axiomatic Basis for Computer Programming
Proceedings of Symposium on Applied Mathematics, Vol. 19 (1967)

E. W. Dijkstra
The Humble Programmer
Communications of the ACM **15** (10), 972: pp. 859–866 *(1972)*

Bibliographie historique (2)

P. Landin
The Next 700 Programming Languages
ACM Programming Languages and Pragmatics Conference, California, 1965.

D. Scott
Data Types as Lattices
Oxford University Monograph, https://www.cs.ox.ac.uk/files/3287/PRG05.pdf

R. Milner
Logic for Computable Functions: Description of a Machine Implementation
Stanford University Press, 1972.

P. Cousot and R. Cousot
Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximations of Fixpoints
Proc. Principles of Programming Languages (POPL), 2007.

ACM 2007 Turing Award
Edmund Clarke, Allen Emerson, and Joseph Sifakis
Model Checking: Algorithmic Verification and Debugging
Communications of the ACM, vol. 52, n. 11, 2009.