Séance du 17 janvier 2023

Conférence-débat de Gérard Berry
avec Marko Erman

# LE LOGICIEL, SA CONSTRUCTION ET SES BUGS

Pourquoi les logiciels peuvent-ils être « buggés » ? Pourrait-on les concevoir de manière à ce qu'ils ne le soient pas ? Le sujet touche aujourd'hui tout le monde puisque, informaticien.ne ou pas, nous y sommes tous et toutes régulièrement confrontés et que nous aimerions vivement trouver un moyen de ne plus l'être. Or, il suffit de comprendre comment fonctionne un logiciel pour constater la facilité avec laquelle les bugs s'introduisent dans toute application informatique. Ce ne sont pas des erreurs ou des pannes des machines, mais bel et bien des erreurs des hommes qui les programment. Ils peuvent perturber ou anéantir le fonctionnement prévu, mais aussi, de façon plus sournoise, compromettre la cybersécurité en ouvrant des vulnérabilités permettant l'attaque des systèmes par les hackers, les mafias, ou même des États. La chasse au bug est donc une activité centrale en informatique, et elle commence bien sûr par l'art d'en faire (beaucoup) moins lors de la conception et de l'écriture des programmes. Pour les éviter, les détecter et les corriger, voire les éradiquer, les méthodes pragmatiques et scientifiques vont des tests bien conduits, qui permettent de trouver des erreurs - mais pas de montrer leur absence -, à la vérification formelle, qui permet de prouver sur papier ou de préférence en machine l'absence de (certains) bugs à l'aide d'algorithmes et de formalismes logiques. Une autre solution, c'est d'intervenir en amont dans le processus de création de programme, par exemple en fournissant aux informaticiens de meilleurs outils de conception, de meilleurs langages de programmation et de meilleurs outils de test et de vérification formelle… « Work in progress » !

**Gérard Berry.** Polytechnicien, ingénieur général du Corps des mines et docteur en mathématiques. Spécialiste des secteurs dits « critiques » de l'aérospatiale, la défense, le transport ferroviaire et l'énergie. Ancien chercheur à l'INRIA et à l'École des mines. Co-fondateur et directeur scientifique de la société Esterel Technologies, principal fournisseur mondial d'outils de conception, de validation et de génération de code pour les applications critiques certifiées fondées sur une description formelle sous forme de modèle dans un langage mathématiquement précis. Professeur émérite au Collège de France, membre de l'Académie des sciences et de l'Académie des technologies.

**Marko Erman.** Ingénieur de l'École Polytechnique, diplômé de Télécom Paris et docteur en sciences. Ancien responsable du management de la recherche au sein de Philips, puis directeur dans le domaine des composants optoélectroniques et des systèmes optiques chez Alcatel, et directeur technique et directeur de la stratégie d'Alcatel Optronics. Aujourd'hui directeur scientifique à Thales et membre de l'Académie des technologies.

# Exposé de Gérard Berry

## *Une science de construction très différente des sciences naturelles*

Au XXe siècle, la « science », c'est-à-dire les sciences de la nature, était l'étude du triangle Matière-Énergie-Ondes. Toute l'industrie, et l'essentiel de la science du XXe siècle ont reposé là-dessus. Information et Algorithmes sont arrivés plus récemment - même si le mot « algorithme » date de 1240 ! -. Mais ils ont changé complètement la donne, même si, bien sûr, il faut de la matière, de l'énergie et des ondes pour exécuter les programmes, très peu en fait -.

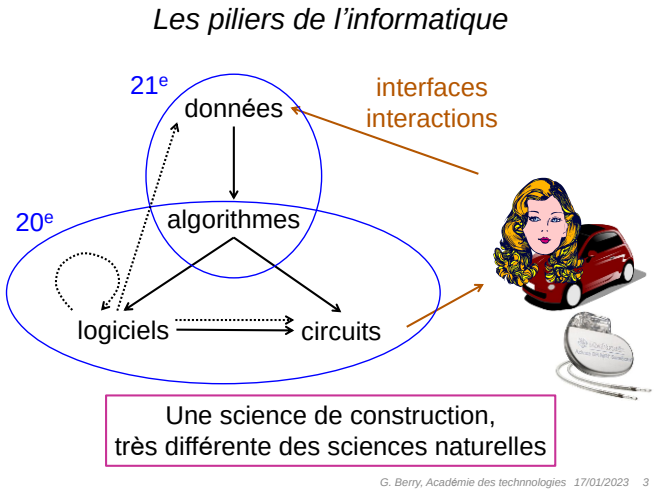
La très grande différence avec les sciences naturelles, c'est qu'en informatique, on construit tout soi-même, un peu comme en mathématiques. C'est nous-mêmes qui fabriquons les objets avec lesquels nous travaillons.

L'autre spécificité de cette science - trop souvent ignorée -, c'est son universalité. Il n'existe qu'une seule sorte d'ordinateur. Le super-ordinateur ou le contrôleur de la machine à laver ne sont conceptuellement qu'une seule et même machine, ne différant que par leur puissance de calcul. En physique, c'est totalement différent : par exemple, un moteur électrique n'a rien à voir avec un moteur à essence… Si les applications sont extrêmement variées, les logiciels eux sont tous de la même nature fondamentale. Et même s'il peut évidemment y avoir des façons de faire un peu différentes, ils requièrent la même culture. Autrement dit, quel que soit le domaine dans lequel on fait de l'informatique, il faut avoir fait les mêmes études. Le problème, c'est que beaucoup de gens parlent d'informatique sans les avoir faites, en particulier du côté des donneurs d'ordres, ce qui engendre bien des problèmes.

## *Les logiciels sont partout et rarement simples*

Les informaticiens comptent 4 piliers de l'informatique, moi j'en cite toujours 5 : données, algorithmes, logiciels, circuits, mais aussi interfaces et interactions homme-machine (figure 1).

Figure 1



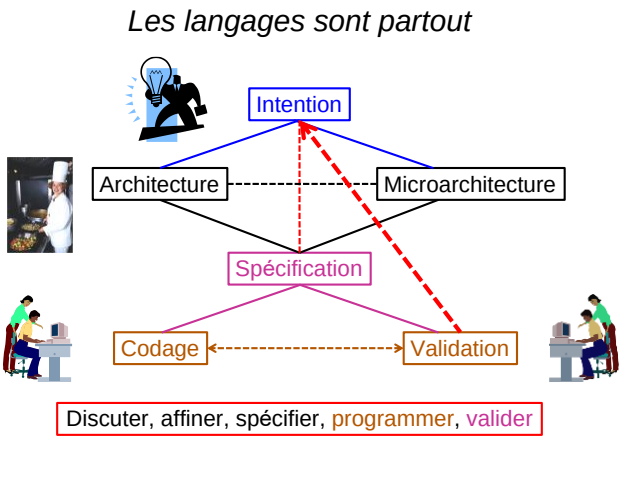

Aujourd'hui, même si les choses vont moins vite qu'on l'imaginait (notamment avec la sécurité, l'informatique embarquée, l'IoT, etc.), les logiciels sont partout : les circuits sont entièrement conçus de façon logicielle, les données sont acquises par prétraitements logiciels, les algorithmes n'ont d'utilité informatique qu'une fois traduits en logiciels, les logiciels sont eux-mêmes fabriqués très souvent par des logiciels… Et ils ne sont pas simples. Ils combinent fréquemment beaucoup de fonctions, leur taille est souvent considérable, et ils n'ont plus aucune existence matérielle. On ne peut plus aujourd'hui imprimer le teste immense des logiciels, ils ne sont visibles que par bouts sur les écrans, ce qui pose un problème inquiétant de pérennité.

Autre particularité de la machine informatique : ce n'est jamais un logiciel qui commet une erreur, car les circuits font exactement ce qu'on leur dit de faire. Un bug vient la plupart du temps d'erreurs de ses concepteurs, ou encore d'erreurs de leurs outils, dont les logiciels qui le traduisent en code machine et ceux qui gèrent la machine. On le voit de façon majeure dans les trous de sécurité. Il peut aussi y avoir une erreur de design de la machine, ce qui est plus rare, mais peut arriver. Donc, l'erreur est toujours humaine !

## *Autant de langages que de fonctions*

Pour écrire un logiciel il faut un langage (figure 2). Un projet, en effet, démarre par une intention qui, une fois qu'elle est traduite en langage machine. En termes de circuits, les architectes font la construction globale et le partage des fonctions et des performances ; les micro-architectes définissent les composants, ce qu'on appelle des IPs pour « Intellectual properties » dans ce domaine, même si le circuit final est matériel. Pour le logiciel, on parle de spécification, codage et validation. L'objectif est que tout l'ensemble finisse par être en phase, ce qui ne va pas de soi.

Figure 2




Pour réussir à formaliser des intentions, les communiquer, et faire en sorte d'aller jusqu'à leur traduction en machine, plusieurs langages de spécification et de programmation sont nécessaires. Si à l'époque de Fortran et Cobol, on pouvait penser que les langages étaient similaires et qu'il suffisait d'en savoir un pour les savoir tous, ce n'est plus du tout vrai. Les langages de programmation sont aujourd'hui nombreux et bien différents, ce qui rend le domaine complexe.

Entre architectes, il s'agit d'abord d'échanger des idées et des pistes de réalisation via des textes, des formules, des schémas, des tableurs, des prototypes… La formalisation précise est difficile et lente, donc rare, surtout dans les projets créatifs et innovants.

Ensuite, on « discute » entre architectes et réalisateurs (programmeurs, designers de circuits), et la spécification nécessite des documents synthétiques et des listes d'exigences (« requirements »), ce qui est très dur à réaliser parce que la machine n'étant pas intelligente, il faut tout dire précisément…

D'autres langages et méthodes sont requis pour critiquer le design, exprimer des retours d'expérience, détecter les incohérences… Il y a là une grosse source de bugs !

Mais il va falloir aussi affiner les spécifications en explicitant les choix de réalisation et en supprimant les ambiguïtés… Mais elles sont souvent de grande taille, et on n'a pas encore les bons langages pour cela. C'est vraiment un gros point faible, et un autre nid à bugs.

Puis il faut programmer, c'est-à-dire rendre l'ensemble exécutable par une machine. Cela nécessite un énorme outillage.

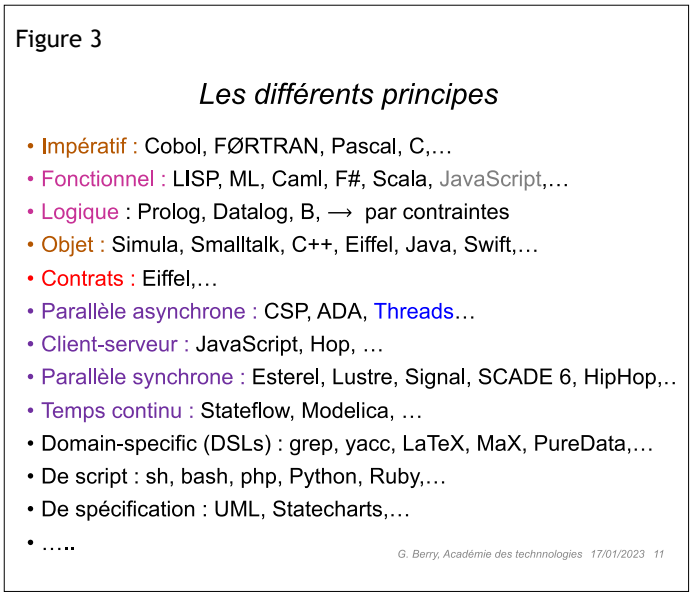
Enfin, vient la validation (test ou preuve) qui demande encore d'autres langages. Par exemple, en circuits hardware, il y a le langage E qui est spécifiquement fait pour la génération de tests. Il existe aussi des langages logiques (assertions, preuves, etc.) qu'on retrouve dans la spécification, mais qui peuvent servir aussi en validation.

## Les constituants d'un langage

Tout commence par une syntaxe. Aux deux clans déjà existants, le « graphique », séduisant mais pas très maniable, et le « textuel », assez abscons, s'ajoute désormais le mixte. Le style de programmation compte également beaucoup. Mais le plus important reste la sémantique. Il ne suffit pas d'écrire des programmes, il faut des méthodes pour définir leur sens. La France a été et reste parmi les leaders mondiaux dans ce domaine... Vient ensuite l'outillage (éditeurs, compilateurs, débogueurs), qui demande beaucoup de travail. Et enfin, les utilisateurs que l'on peut répartir en deux clans eux aussi : les adorateurs et les contempteurs. On le voit, le paysage n'est pas simple.

## Les différents principes

Figure 3



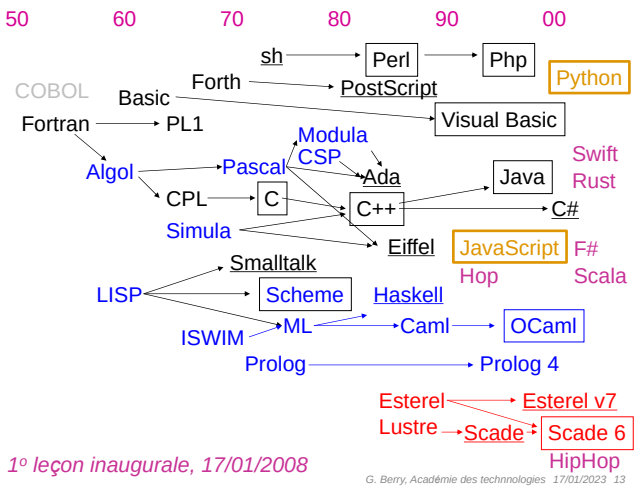

## Pourquoi autant de langages ?

Parce que personne ne sait tout faire et que les besoins sont difficiles à combiner. En outre, la différence est très forte entre, d'un côté, les machines de von Neuman qui sont le modèle de base des circuits et, de l'autre, le lamba-calcul de Church, un formalisme qui fonde de plus en plus les langages de programmation (ils ont équivalents mais techniquement très différents). Idem pour la logique, qui est encore autre chose, ou les blocks-diagrams des automaticiens... De plus, les domaines d'utilisation ont des vocabulaires internes complètement différents, car il y a beaucoup de types d'applications. Ajoutons les contraintes universelles de souplesse, efficacité et sûreté. On sait très rarement

mettre tout cela ensemble. Enfin intervient aussi le type de machine et de langage qu'on utilise : séquentielle ? Parallèle ? Code interprété ou compilé ? Langage typé ou pas ? À l'intérieur de tout cela, il faut faire sa place. Une place qui, même si elle peut être parfois dictée par la mode, devrait plutôt l'être par le problème abordé.

En 2008, dans ma première leçon inaugurale, j'avais donné une petite cartographie des grands groupes de langages (figure 4) :

Figure 4



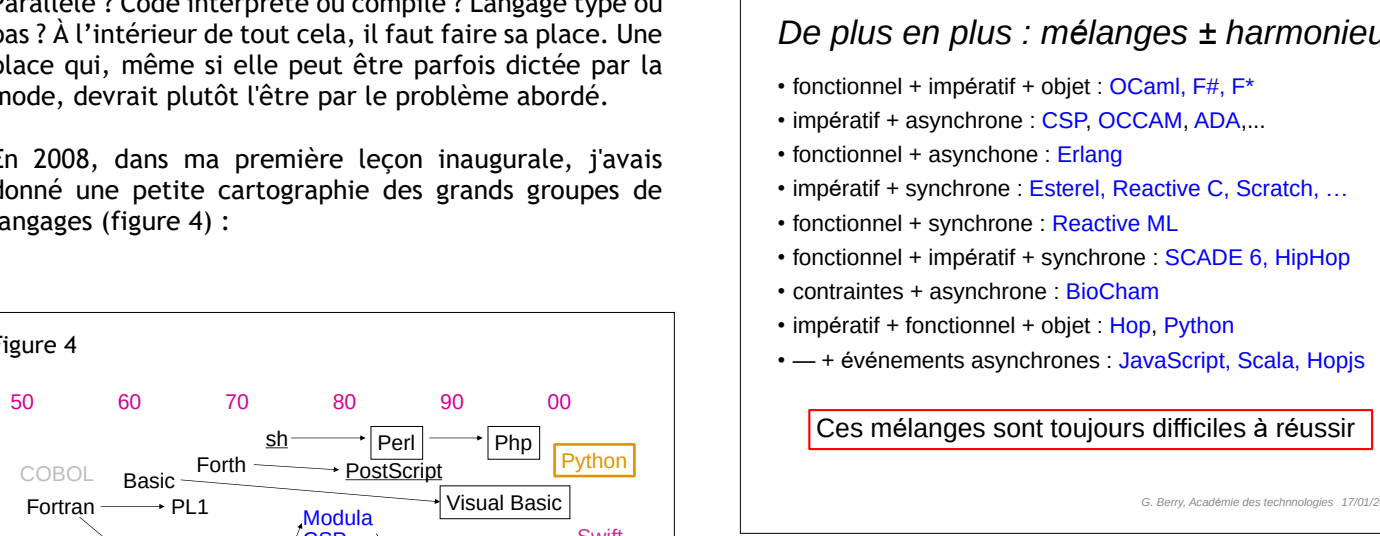

D'autres sont arrivés ensuite et ont pris le dessus. Par exemple, Scala qui est un langage très intéressant, Rust qui est de plus en plus utilisé et a été promu dans Linux parce qu'il permet d'éviter les erreurs qu'on commet si facilement en C et qui sont à l'origine de gros trous de sécurité… Et puis, à l'heure actuelle, les deux langages leaders que sont JavaScript et Python, qui n'ont pas que des qualités.

## *Les guerres de religion*

« L'*impératif* est un nid à bugs à cause des pointeurs et de la gestion de la mémoire par l'utilisateur. » Reproche classique du *fonctionnel*. « Mais non, rétorque *l'impératif*, c'est très naturel, il y a qu'à faire comme la machine, on suit avec les doigts, et il suffit d'ajouter ce qu'il faut, les assertions logiques, les contrats, l'analyse statique… » Ou encore : « Le fonctionnel, c'est pour les intellos, c'est inefficace. »

Cela fait quarante ans que la querelle dure, elle ne va pas s'arrêter de sitôt. Alors aujourd'hui, on essaie de faire des mélanges. Certains marchent bien, d'autres pas du tout (figure 5).

Figure 5



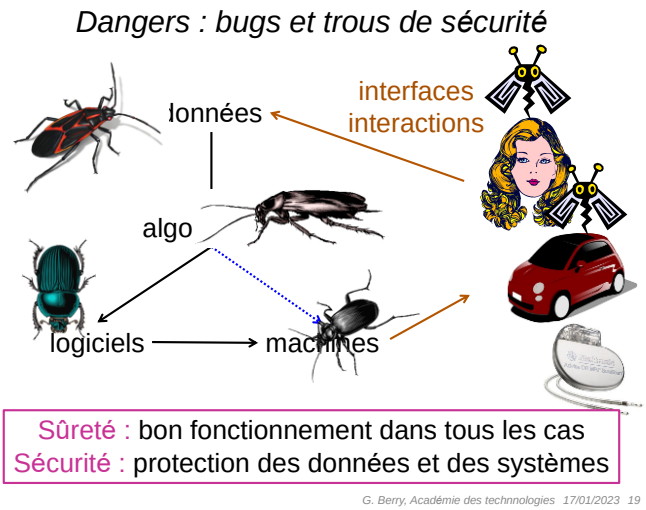

## *Les raisons du succès (ou de l'échec) d'un logiciel*

- Un design harmonieux, une sémantique claire.
- De bonnes documentations et implémentations : on doit pouvoir comprendre ce que font les programmes, si possible sans avoir à entrer dans la machine…
- De bons environnements de programmation.
- Une liaison fine avec le test et la vérification formelle.
- Une bonne adaptation à un type de problème (celui qu'on a envie de résoudre).
- Une communauté active d'utilisateurs.
- Une installation triviale : le gros défaut des logiciels libres vient souvent d'installations très compliquée qui déroutent les utilisateurs.
- Des bons outils de formation.
- Un bon nom, pour le succès médiatique (Java, c'était très bien, Esterel a eu aussi pas mal de succès).
- Enfin et surtout, un bon logiciel doit apporter une solution à un problème que les autres ne savent pas résoudre. Java, c'était le téléchargement, JavaScript, le web. SCADE 6, c'était le contrôle temps réel critique, comme ses ancêtres Esterel et Lustre. Pour le « scripting », c'est tcl, perl, Ruby, Python… Pour l'enseignement, c'est aujourd'hui Scratch et Python.

## *Dangers : bugs et trous de sécurité*

Aujourd'hui la situation de la sûreté (correction du comportement) n'est pas très bonne, quant à celle de la sécurité (résistance aux attaques), elle est mauvaise (figure 6). Et contrairement aux idées reçues, la faiblesse de la sécurité, ce sont les programmes. Prenons un exemple de faille typique. On est dans société financière où le programme de gestion démarrait toujours très bien le matin, sauf un jour par semaine. Pourquoi ? Il a fallu du temps pour trouver le bug. La doc du système disait : au démarrage, dans une zone de huit caractères, on écrit le nom du jour. Donc ils avaient réservé huit caractères, et ensuite ils tapaient un mot, ou un y pour démarrer en mode maintenance, et sinon rien. Que s'est-il donc passé ? Le lundi (« monday »), pas de problème, le mardi (« tuesday ») non plus. Mais le mercredi, le Y de « wednesda-y », qui a 9 caractères, a fait croire au système que l'utilisateur voulait démarrer en mode maintenance. C'est ce qu'on appelle une corruption mémoire. Autant trouver une aiguille dans une botte de foin !

Figure 6




Des bugs de ce genre, on en trouve entre 5 et 30 dans chacune des fréquentes releases de Macosx, de Windows et même de Linux... Ceux qu'on peut lister dans les rapports de mise à jour sont ceux qui ont été trouvés à l'extérieur de l'entreprise par des extérieurs qui se font payer pour ça. Il y en a probablement bien plus.

Tout cela est dû à des pratiques de programmation trop orientées vers la seule efficacité, sans suffisamment de contraintes de correction, et au manque de compétence des donneurs d'ordre, notamment pour la sécurité... Tant qu'il n'y avait pas d'attaque, ce n'était pas grave. Aujourd'hui, c'est très grave mais il est trop tard parce que ces logiciels sont souvent trop difficiles à relire et donc à corriger.

## *Quand le bug fait des morts*

L'absence de règles de certification pertinentes peut conduire à des situations d'extrême dangerosité. Certes, pas dans le monde de l'avionique, particulièrement sérieux. Le crash du 737 Max n'était pas dû à un bug de logiciel mais à un bug de design global du système... En revanche, si l'on prend l'exemple des voitures, aujourd'hui équipées d'une quantité d'informatique sidérante, il n'est pas difficile de rencontrer des bugs pas toujours anodins.

Comme chez Toyota en 2015, où une défaillance du logiciel du contrôle moteur a fait 89 morts, et coûté plus de 2,5 milliards de dollars d'amende à l'entreprise, et plusieurs milliards à la suite des class actions. Que se passait-il ? Les voitures partaient à fond, même avec le frein à main activé, sans aucun moyen de les arrêter. Le crash total. Un rapport d'expert de 750 pages explique en substance que « le logiciel est beaucoup trop complexe », que « beaucoup de choses ne sont pas testables », qu'il n'y a « aucun moyen de fabriquer une suite de tests », et que « mieux vaut ne pas y toucher sous peine de rajouter des bugs… ». « L'ensemble de l'architecture de sécurité est un château de cartes ». Dans ce cas de Toyota, ce sont d'importants dépassements et écrasements de mémoire qui sont à l'origine des crashs, montrant une ignorance curieuse pour la première industrie du monde en automobile...

## *La chasse aux bugs*

Le premier outil pour prévenir le bug est le test, notamment le test aléatoire : le projet Csmith aux États-Unis a trouvé des centaines de bugs sur les compilateurs C libres ou industriels - sauf sur CompCert, dont nous reparlerons plus loin. Comme le soulignait Dijkstra, déjà en 1973, un test permet de trouver des bugs, « mais pas de montrer qu'il n'y en a pas ».

Que peut-on faire de plus ? Par exemple, instrumenter les programmes avec des assertions logiques qu'on peut prouver. C'est Turing qui a, le premier, fait une preuve de programme en l'instrumentant, en 1947.

Il existe aussi des contrats, qui sont des prédicats plus généraux (B, Coq…) Et puis des moteurs automatiques qui permettent d'automatiser les calculs : les Binary Decision Diagrams, par exemple, qui permettent de faire du calcul booléen efficace, et désormais SAT qui a des applications fantastiques, par exemple en circuits.

L'autre méthode pour « sécuriser » les logiciels, c'est la preuve mathématique en machine. Robin Milner, en 1972, avait créé le premier système. A suivi une grande révolution théorique avec la théorie des constructions de Thierry Coquand et Gérard Huet à l'INRIA, qui a donné le système Coq, l'un des trois grands systèmes de preuves actuels, peut-être le plus abouti, celui avec lequel a été formellement développé et prouvé correct

le compilateur CompCert précité, par Xavier Leroy et son équipe à l'Inria…

À noter que la France a été pionnière en la matière, puisque le premier système industriel non testé mais vérifié par vérification formelle a été celui du RER A (par Jean-Raymond Abrial et son équipe) !

### *Perspectives pour demain*

Nos points faibles aujourd'hui sont : le manque de formation et l'excès de confiance de l'ensemble de la chaîne (on fait vite, on teste un peu, on met en service dès que possible, on attend les retours et on corrige…), l'ignorance des clients qui ne connaissent souvent rien du sujet, et le fait que les logiciels vieillissent vite…

On peut et on doit faire mieux, et la recherche peut et doit aider. Il est urgent d'agir notamment dans les domaines où l'enjeu peut être grave, comme les hôpitaux.

## Débats

***On a beaucoup parlé du bug de l'an 2000. Toute l'industrie et quelques autres se sont mobilisés pour éviter le pire. Et finalement, ça s'est passé plutôt mieux qu'on craignait. Quelle leçon peut-on en tirer ? Peut-on craindre, à l'avenir, d'autres problèmes systémiques de ce genre ?***

Le bug de l'an 2000 n'était pas du tout un bug, c'était un choix, celui de coder la date sur une taille trop petite qui risquait de déborder. Mais les logiciels de l'époque étaient extrêmement difficiles à réparer. Changer le format du temps aujourd'hui n'aurait pas le même effet. À l'époque, les programmes étaient tous faits en dilettante, surtout les plus anciens, et il a fallu fournir un effort énorme. C'était aussi l'occasion de mieux les refaire… À l'heure actuelle, le gros problème ce sont les accès-mémoire illégaux dans les systèmes et la sécurité. C'est tellement facile à trouver pour les hackers qui les achètent sur le dark web à partir de 50 €… Et il y a aujourd'hui tellement de logiciels « buggés » on ne sait pas où... C'est un énorme problème. Nombre de gens chez Apple et Microsoft doivent être en train de scruter le système d'exploitation pour essayer de trouver les bugs et de les patcher. S'il y en a autant, c'est parce que personne auparavant ne testait assez, c'était trop cher, cela ralentissait le système. C'était la grande difficulté… Mais il y en aura moins désormais. Un langage comme Rust justement est conçu pour programmer de manière efficace mais sans cet inconvénient.

***Que faut-il faire pour mieux former les développeurs ? Y a-t-il un pays modèle, par exemple l'Inde, où la formation serait significativement meilleure en informatique que chez nous, et qui pourrait nous inspirer ? Que pensez-vous de l'initiative type Ecole 42 ?***

En France, l'idée de former les gens à l'informatique a été longtemps catégoriquement refusée. Au lycée, c'est arrivé en 2019, mais sans les profs, et sans qu'aucun de ces cours ne s'appelle « informatique ». Les filles sont massivement détournées de cette branche, et elles y sont de moins en moins nombreuses dans les études supérieures. En revanche, en Tunisie, en Algérie ou au Maroc, la première matière est la médecine, et l'informatique arrive juste après. Le taux de filles en informatique est de 60 % en Tunisie et de plus de 50 % en Algérie. Nous, sommes restés au Moyen Âge...

Les écoles 42, c'est très différent, cela va donner de l'informatique plutôt bas de gamme, mais souvent utile. Les cours ne sont pas brillants, et ils forment des gens qui vont faire les sites web de tout le monde. Cela dit, la pénurie de gens formés est telle que c'est bien. L'école 42, en fait, joue aussi un rôle que refuse de remplir l'Éducation nationale en recrutant au sein de publics différents. Pour les profs, il a fallu de longues bagarres pour créer des cours au lycée, un CAPES et une agrégation. C'est fait, mais il n'y a pas de filles. En Inde, l'informatique est la matière numéro un. Et il y a des filles ! Même chose en Chine. Chez nous, ça ne progresse encore que très lentement.

***Est-ce que l'Open Source est porteur de plus de sûreté et de sécurité ?***

Souvent oui, tout simplement parce que les gens qui l'écrivent connaissent le problème et le niveau d'exigence — quand on parle de vrais projets, du genre Linux —. Leurs programmes sont souvent beaucoup plus sûrs parce qu'ils ne travaillent pas avec le même rythme ni avec les mêmes contraintes. Ils prennent le temps qu'il faut et ne se font pas payer au rendement.

***Étant donnée la place prise par le digital dans la vie quotidienne, ne faudrait-il pas imposer par la réglementation que tous les logiciels soient testés avec des preuves formelles ?***

Non, ce n'est pas possible, on n'a pas les gens pour. Faire des preuves formelles de logiciel ne demande pas le

même niveau que d'écrire un programme. En sécurité par exemple, il existe des sociétés qui savent faire ça : Prove&Run en France, par exemple, comme son nom l'indique… Mais on ne peut pas le faire pour tout, il s'écrit tellement de millions de lignes de code tout le temps, c'est impossible. On peut déjà essayer d'être moins « bête », tester par exemple des dépassements de tableaux au lieu de les laisser se faire ! Ce n'est pas très compliqué, mais une fois que les programmes sont écrits, c'est dix fois plus difficile. Donc il faut investir.

### *Comment responsabiliser les industriels et les pousser à un arbitrage sûreté-sécurité/ innovation plus favorable à la sûreté sécurité ?*

Un modèle qui responsabilise tout seul, en ce moment, ce sont les attaques. L'attaque qui a fait perdre 200 millions d'euros à Saint-Gobain a modifié la perception du conseil d'administration de la société et de son président… Sur le plan législatif, l'ANSSI fait un travail absolument remarquable. Mais faut-il obliger les gens à respecter des procédures de sécurité, de même que les assurances obligent leurs adhérents à sécuriser l'entrée dans les bâtiments ? Pour les bugs, la question concerne le conseil d'administration. Mais combien de personnes, dans ces conseils-là, savent ce qu'est un logiciel ? Il est nécessaire qu'il y ait des personnes compétentes sur ces problèmes dans les organes de décision.

**Mots clés :** bug, codage, langage informatique, logiciel, ordinateur, programmation, sécurité informatique, sûreté informatique

**Citation :** Gérard Berry & Marko Erman. (2023). *Le logiciel, sa construction et ses bugs*. Les soirées de l'Académie des technologies.@

**Retrouvez les autres parutions de l'Académie des technologies sur notre site**

**Académie des technologies. Le Ponant, 19 rue Leblanc, 75015 Paris. 01 53 85 44 44. academie-technologies.fr**

**Production du comité des travaux. Directeur de la publication : Denis Ranque. Rédacteur en chef de la série : Hélène Louvel. Auteurs : Marie-Claude Treglia. N°ISSN : en attente.**

**Les propos retranscrits ici ne constituent pas une position de l'Académie des technologies et ils ne relèvent pas, à sa connaissance, de liens d'intérêts. Chaque intervenant a validé la transcription de sa contribution, les autres participants (questions posées) ne sont pas cités nominativement pour favoriser la liberté des échanges.**