ACADÉMIE DES TECHNOLOGIES
POUR UN PROGRÈS RAISONNÉ, CHOISI ET PARTAGÉ

LA PROGRAMMATION : INDUSTRIE DES IDÉES PURES

Séance thématique organisée par Bertrand MEYER
Membre fondateur de l'Académie des technologies
Coordinateur de l'ouvrage *The French School of Programming*

Séance du 11 juin 2025

Résumé

Au cœur de tous les processus du monde moderne se trouvent des programmes informatiques, souvent complexes, dont la qualité est critique pour le bon fonctionnement de la société. Depuis une soixantaine d'années, la programmation et sa généralisation au « génie logiciel » se sont affirmées comme une discipline scientifique autonome et exigeante, avec ses méthodes propres, ses paradigmes, ses théorèmes et ses défis. Les chercheurs et industriels français ont joué un rôle important dans l'histoire de la discipline et contribuent activement à son avenir. Le livre *The French School of Programming* (2024) réunit les contributions de quelques-uns des chercheurs français les plus prestigieux dans le domaine. À l'occasion de sa sortie, l'Académie des Technologies a invité les auteurs à présenter à un large public technophile, mais non nécessairement informaticien, les principaux problèmes et solutions de la programmation et du génie logiciel. Après un « portrait-robot » du programmeur et une présentation générale de l'ouvrage, trois interventions illustrent quelques-uns des thèmes abordés.

Intervenants

Gérard BERRY
Professeur émérite au Collège de France
Membre de l'Académie des technologies

Marie-Claude GAUDEL
Professeur émérite d'informatique
à l'Université Paris-Sud

Jean-Marc JÉZÉQUEL
Professeur de génie logiciel à l'université
de Rennes

Jean-Pierre BRIOT
LIP6 – Sorbonne Université – CNRS

Sommaire

La programmation ou l'industrie des idées pures : défis et paradigmes 2
Présentation de *The French School of Programming* 6
Le test peut aussi être formel (trente ans après) 7
La modélisation en logiciel, des outils CASE à l'apprentissage machine 8
Des procédures, objets, acteurs, composants et services aux agents 9

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.

La programmation ou l'industrie des idées pures : défis et paradigmes

Bertrand Meyer

Le célèbre tableau de René Magritte, *La trahison des images*, montrant une pipe accompagnée du texte « Ceci n'est pas une pipe », souligne la différence entre un objet et sa représentation. En programmation, cette différence est souvent difficile à isoler car un programme est à la fois une description et une réalisation, et les mécanismes utilisés pour les deux tâches (« spécification » et « implémentation ») sont remarquablement similaires. Cette situation est unique en ingénierie : personne ne confondrait un plan d'architecte avec un bâtiment, ou un diagramme de circuit avec le circuit. En programmation elle est omniprésente et complique le raisonnement : tout élément de logiciel est (sauf au niveau le plus abstrait) l'implémentation d'un autre et (sauf au niveau le plus concret) la spécification d'un autre. Nous sommes dans l'industrie des idées pures.

Une discipline méconnue

Chacun a quelques notions des grands paradigmes de la physique, de la biologie, des mathématiques ou de l'épistémologie, tels que la théorie de la relativité, le principe d'incertitude en mécanique quantique, le chat de Schrödinger, la théorie de l'évolution en biologie, le paradoxe de Russell, le théorème d'incomplétude de Gödel, la notion de falsifiabilité de Popper ou la rupture épistémologique de Bachelard. En revanche, même le grand public cultivé scientifiquement ignore, en général, quels sont les enjeux intellectuels de la programmation. Je vais essayer de vous les faire partager en vous présentant différentes facettes du personnage du programmeur.

Le programmeur est Gulliver

Comme Gulliver, le programmeur est confronté à des échelles tantôt microscopiques et tantôt infinies. Accéder à une information conservée dans la mémoire centrale d'un ordinateur de 64 GB prend 1 nanoseconde ; si elle est stockée sur un disque, il faut 10 millisecondes, c'est-à-dire 10 millions de fois plus ! La réponse à une requête ChatGPT nécessite entre 100 milliards et 10 billions d'instructions de base, qu'un ordinateur moderne exécute au rythme de centaines de milliards par seconde.

Imaginons que, dans le système d'exploitation Windows 11, qui comprend environ 60 millions de lignes de code, soit 10 milliards d'éléments unitaires d'information (« bits »), on modifie un seul d'entre eux, en remplaçant un 0 par un 1. Il est possible que cette substitution reste totalement inaperçue pendant des années ou, au contraire, que tout le programme cesse subitement de fonctionner. Voilà pourquoi les programmeurs semblent parfois un peu fous. Par comparaison, l'Airbus A 380, l'un des objets physiques (par opposition aux systèmes logiciels qui sont virtuels) les plus complexes réalisés par l'homme, comprend « seulement » 4 millions de composants.

Pour s'assurer du bon fonctionnement d'un programme, il faut se doter de techniques de gestion très rigoureuses, car personne n'est capable d'avoir une vision à la fois générale et précise de ces millions de lignes de code, et la moindre erreur peut s'avérer grave.

Une vedette

Un programmeur est une « vedette » : tout dans le monde d'aujourd'hui repose sur l'informatique. Selon la formule de Marc Andreessen, le co-développeur de Mosaic, « *Software is eating the world* ». L'industrie de l'information (en excluant les télécoms) représente 3,7 trillons de dollars dans le monde, dont 70 milliards d'euros en France. Elle mobilise 20 millions de programmeurs dans le monde, dont 600 000 en France.

Un acrobate

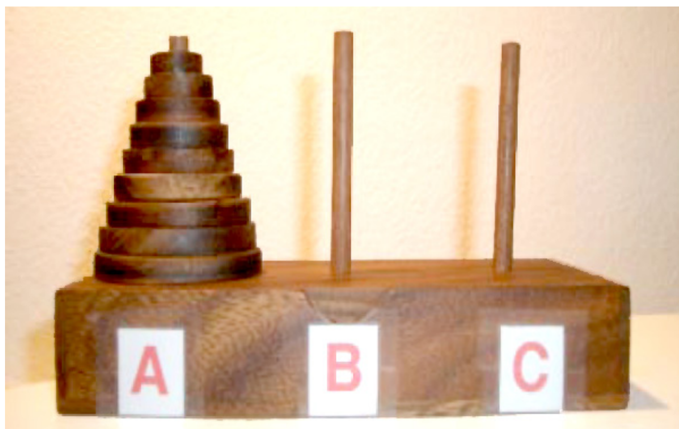
Le programmeur est également un acrobate, car il est toujours en danger de chuter. Dès 1936, avant même la fabrication du premier ordinateur, Alan Turing avait l'intuition de ce que pourraient ou ne pourraient pas faire ces futures machines. Il affirmait « qu'il n'y a pas de procédure mécanique générale pour déterminer si un programme s'arrêtera », le terme *mécanique* signifiant *automatisable*. Pour démontrer cette affirmation, il faut recourir à ce que l'on appelle une boucle, par exemple *from i : = 0 until i> N loop i : = i + 2 end*, ce qui signifie que l'on va exécuter l'action 0 une ou plusieurs fois jusqu'à ce que la condition de fin soit vraie. En l'occurrence, la condition de fin est que le programme se termine. Si je range ce programme dans le même fichier qu'un autre programme, selon lequel il doit exécuter l'action jusqu'à ce que la propriété « se termine » soit fausse, on tombe sur une variante des grands paradoxes mathématiques du début du 20e siècle (de Russell à Gödel) : l'indécidabilité de la terminaison des programmes, démontrée par Turing en 1936 Si l'un des programmes se termine, l'autre ne se terminera pas, car la boucle se poursuivra indéfiniment, et inversement.

Au-delà de cette question, le théorème de Rice affirme que tout problème intéressant dans le domaine de la sémantique des programmes est indécidable, au sens où il n'existe pas, pour le résoudre, de procédure générale automatique. De fait, les problèmes indécidables sont le quotidien du programmeur. Par exemple, quand on demande à un outil de vérification de logiciel de déterminer si un autre programme fonctionne ou non, il n'est pas capable de garantir dans tous les cas une réponse par oui ou par non. Il y aura toujours des cas où la réponse ne peut être que « je ne sais pas. » Cette indécidabilité quotidienne devrait rendre le programmeur modeste, si toutefois les termes *programmeur* et *modeste* étaient compatibles.

Un tricheur

Un programmeur peut aussi être comparé à un joueur de bonneteau. Il triche en permanence, un peu comme les enfants, qui emploient volontiers le conditionnel (« *Toi, tu serais un cowboy et moi, je serais un indien* »). Quand un programmeur ne sait pas quelque chose, il le suppose. Cela se vérifie, par exemple, à propos de la fonction récursive, illustrée par l'image figurant sur les boîtes de Vache qui Rit : une vache qui porte aux oreilles deux boîtes de Vache qui Rit, chacune portant l'image d'une vache qui rit et porte aux oreilles deux boîtes de Vache qui Rit…

Le jeu des tours de Hanoï a été inventé par le mathématicien français Édouard Lucas en 1883. Ce casse-tête mathématique considère trois aiguilles sur lesquels peuvent être empilés n disques de tailles variées ($n = 64$). Au départ, ils sont tous sur l'aiguille A. Le but du jeu est de les *transférer* sur l'aiguille C en utilisant l'aiguille B comme intermédiaire et en respectant deux contraintes : on ne peut déplacer qu'un disque à la fois ; et il est interdit de placer un disque sur un autre de diamètre inférieur. On voit facilement que le nombre de déplacements minimum est de $2^n - 1$ L'algorithme dit : « *transférer* » $n - 1$ disques de A vers B ; déplacer le disque restant (le plus grand) de A vers C ; « *transférer* » $n - 1$ disques de B vers C. Le terme « *transférer* » représente l'algorithme lui-même, appliqué à $n - 1$ disques sur les n d'origine (à distinguer de « déplacer » qui dénote l'opération élémentaire consistant à prendre le disque au sommet d'une pile pour le mettre sur une autre). Cet exemple offre une illustration de la puissance et de la lisibilité des programmes définis de façon récursive.



La tour de Hanoi ($n = 9$)

Un autre programme récursif est celui qui permet de calculer la suite de Fibonacci, du nom du mathématicien italien qui, en 1202, avait proposé de prévoir la croissance d'une population de lapins selon des règles prédéfinies : au début du premier mois, il n'y a qu'un couple de lapereaux ; ils ne peuvent procréer qu'à l'âge de deux mois ; chaque début de mois, toute paire susceptible de procréer engendre exactement une nouvelle paire de lapereaux ; les lapins ne meurent jamais. On peut l'exprimer par la formule suivante : $fib(n) = fib(n-1) + fib(n-2)$ *for* $n > 1$ et la représenter sous la forme d'un arbre. Gérard Berry, dans un article de 1976, *Calculs ascendants des programmes récursifs*, a montré qu'il était possible de procéder aux calculs de bas en haut de cet arbre, à partir d'une fonction partielle (c'est-à-dire définie pour quelques arguments seulement, par exemple pour des valeurs déjà connues, comme 0 et 1), que l'on étend progressivement.

Les programmes récursifs sont un exemple frappant de la méthode consistant à « faire semblant » (supposer que quelque chose est possible, et s'en servir), universelle en programmation.

Un démiurge

Quand un programmeur ne sait pas faire quelque chose, il le suppose, et quand il ne dispose pas de l'élément dont il a besoin, il le construit. On peut donc le comparer également à un démiurge qui fabrique son golem, façonné afin d'assister son créateur. Encore faut-il être en mesure de vérifier qu'il effectue bien ce que son maître lui a ordonné.

Un donneur d'ordres

Le programmeur est donc aussi un donneur d'ordres. Une partie fondamentale de son travail est ce que l'on appelle l'ingénierie des exigences *(requirements engineering*), exigences qui peuvent être informelles (c'est-à-dire définies en langage naturel ou à travers des diagrammes) ou formelles (c'est-à-dire exprimées dans une notation mathématique aux propriétés rigoureusement définies).

Une girouette

Souvent client varie, bien fol qui s'y fie. En principe, apporter un changement à un programme est facile, mais les conséquences de ce changement peuvent être colossales. Si l'architecture du programme n'a pas été conçue de façon à intégrer de futurs changements, des réactions en chaîne peuvent se produire et mener à une catastrophe. Gardant à l'esprit que « la seule constante, c'est le changement », le programmeur doit construire des architectures de logiciel anticipant des modifications ultérieures.

Un gribouille

Le programmeur est aussi un gribouille, c'est-à-dire quelqu'un qui ne cesse de commettre des erreurs, en raison de la complexité des problèmes auxquels il s'attaque et du nombre de détails à prendre en compte. L'échec du vol inaugural d'Ariane 5 en 1996 est lié à un bug du logiciel de navigation, qui a provoqué l'explosion de la fusée au bout de 37 secondes.

Un policier

Sachant qu'il importe non seulement de définir le programme mais d'être en mesure de le vérifier, on peut aussi comparer le programmeur à un policier. Il existe deux types de vérification des logiciels, l'une « tels qu'ils devraient être » et l'autre « tels qu'ils sont ».

La première approche a conduit à des techniques de méthodologie de la programmation telles que la « Conception par Contrat » développée par l'auteur et intégrée dans le langage Eiffel. Il faut cependant dans la plupart des cas vérifier des programmes qui n'ont pas toujours été écrits avec le soin nécessaires : d'où la nécessité de « vérifier les programmes tels qu'ils sont ».

Deux types de techniques existent : dynamiques, qui exécute les programmes sur des données représentatives, et statiques, qui examine simplement le texte du programme.

Les méthodes dynamiques utilisent le test, qui exécute le programme pour trouver des erreurs. Le test exige un état d'esprit bien particulier, celui de Méphistophélès qui, dans l'Acte I de *Faust*, se présente comme « *l'esprit qui toujours nie* ». Comme l'a souligné l'informaticien hollandais Edsger Dijkstra, « un test ne peut pas démontrer que le programme n'a pas d'erreurs, mais seulement qu'il a des erreurs ! » Une des difficultés est de définir les cas de tests qui permettront de détecter les erreurs. Marie-Claude Gaudel, pionnière dans ce domaine, expliquera tout à l'heure comment les méthodes formelles, c'est-à-dire mathématiques, peuvent être appliquées aux tests tout comme aux preuves.

Quant aux méthodes statiques, on peut notamment citer l'interprétation abstraite, basée sur la théorie des connexions de Galois et développée par Patrick et Radhia Cousot, qui fait l'objet d'un des chapitres du livre. Elle consiste à vérifier une version simplifiée mathématiquement du programme, puis à déduire de cette démarche les propriétés du programme lui-même.

Enfin, plusieurs exemples français illustrent la méthode de la preuve complète de programme et ont obtenu une large audience internationale. Il s'agit en particulier de Coq (récemment renommé Rocq) initiée par Thierry Coquand et Gérard Huet ; Frama-C développée par le CEA et Framatome ; la méthode et les outils B de Jean-Raymond Abrial ; et l'environnement de preuve AutoProof pour Eiffel. Cette dernière consiste à construire les programmes de telle façon qu'on puisse leur faire confiance, ce qu'on appelle parfois *correction by construction*.

Un aviateur

La démarche consistant à s'élever par l'abstraction au-dessus de la masse invraisemblable de détails que représente un programme permet de comparer le programmeur à un aviateur. On peut citer à ce sujet un extrait de *La Vie de Henry Brulard*, de Stendhal, à propos d'un mauvais professeur de mathématiques : « *Sans cesse M. Dupuy faisait des phrases emphatiques sur ce sujet* [l'algèbre]*, mais jamais ce mot simple : C'est une division du travail qui produit des prodiges comme toutes les divisions du travail et permet à l'esprit de réunir toutes ses forces sur un seul côté des objets, sur une seule de leurs qualités. Quelle différence pour nous si M. Dupuy nous eût dit : 'Ce fromage est mou ou il est dur ; il est blanc, il est bleu ; il est vieux, il est jeune ; il est à moi, il est à toi ; il est léger ou il est lourd. De tant de qualités ne considérons absolument que le poids. Quel que soit ce poids, appelons-le A. Maintenant, sans plus penser absolument au fromage, appliquons à A tout ce que nous savons des quantités'. Cette chose si simple, personne ne nous la disait dans cette province reculée.* »

En programmation, on peut appliquer cette démarche d'abstraction à un compte bancaire, par exemple, en le considérant comme une théorie mathématique faite d'axiomes et de théorèmes : « On peut retirer du compte le montant m si et seulement si m est inférieur ou égal au solde (« précondition ») ; le solde après l'opération sera égal à l'ancien solde $- m$ ». Cette approche, utilisant les idées de « types abstraits » introduits à l'origine par Barbara Liskov, est à la base de la Conception par Contrats.

Un conservateur

Une des notions fondamentales dans les sciences est celle d'invariant. Il en existe aussi en informatique, ce qui permet de comparer le programmeur à un conservateur. Si l'algorithme conçu par Euclide il y a 2 500 ans fonctionne et permet de calculer le PGCD (plus grand commun diviseur), c'est parce que, pour calculer le PGCD de deux variables m et n, on s'appuie sur une série de soustractions (si n est plus grand que m, on retire n de m, et réciproquement), et qu'une boucle maintient un invariant fondamental : la propriété que le PGCD de deux nombres différents est le même que le PGCD du plus petit d'entre eux et de leur différence. L'autre propriété fondamentale est que le PGCD de deux nombres égaux est leur valeur commune. L'algorithme est entièrement déterminé par ces deux propriétés, tout particulièrement l'invariance exprimée par la première.

Un artificier

Le programmeur est aussi un artificier, comme le savent ceux qui ont cliqué par erreur sur la pièce jointe d'un courrier électronique et ont provoqué une catastrophe. Ces incidents sont le résultat d'une idée géniale développée par Turing et Von Neumann, selon laquelle il est possible de traiter un programme comme une donnée. Cet aspect est développé en particulier par la théorie du lambda-calcul, développée par Church et Curry à partir de années 30 et à la base des langages de programmation dits « fonctionnels ». L'École française de lambda-calcul est particulièrement importante, représentée dans le livre par un article important de Jean-Jacques Lévy.

Un jongleur

On peut également comparer le programmeur à un jongleur, en faisant référence à l'un des domaines fondamentaux de la programmation, le parallélisme. Un téléphone portable assez simple comprend 8 processeurs travaillant en parallèle, et un modèle plus sophistiqué pourra en contenir 16 ou 32. Toute la difficulté est de les synchroniser. Cette question apparaît dans de nombreux cas pratiques : lors de la réservation d'une place d'avion, comment s'assurer que, pendant que vous payez votre place, quelqu'un d'autre ne va pas réserver la même ? L'un des meilleurs spécialistes de ces questions est Michel Raynal, également l'un des auteurs du livre.

Les autres facettes du programmeur

J'aurais pu citer de nombreuses autres facettes du métier de programmeur. C'est bien sûr un *mathématicien*, mais aussi un *linguiste*, puisqu'il passe son temps à se poser des questions de sémantique et de syntaxe ; un *voleur*, qui ne cesse de réutiliser du code en puisant dans les centaines de bibliothèques de logiciels ; un *architecte*, car un logiciel ne peut pas fonctionner s'il n'est pas structuré de façon très rigoureuse.

L'école française d'informatique

Nous avons la chance, en France, de réunir quelques-unes des figures majeures de la programmation, avec des pères fondateurs tels que Marcel-Paul Schützenberger, Jacques Arsac, Maurice Nivat et Claude Pair, mais aussi Gilles Kahn (à qui l'ouvrage est dédié) et (dans un ordre arbitraire, et sans nul doute avec des omissions involontaires) Jean-Raymond Abrial, Véronique Donzeau-Gouge, Joseph Sifakis, Philippe Kahn, Wendy McKay, Radhia Cousot, Christine Paulin-Mohring, Alain Colmerauer, Claire Le Goues, Martin Monperrus, Benoît Baudry, Benoît Combemale, Yves Le Traon, Xavier Leroy, Jean Bézivin, Pierre Cointe, Claude et Hélène Kirchner, Alain Chesnais, Louis Bolliet, Christine Choppy, Bernard Vauquois, Jeanne Poyen, François Genuys, Jean-Claude Boussard, Irène Guessarian, Yann Le Cun, Jean-François Colonna, Pierre Lescanne, Jean-Christophe Fillâtre, Radhia Cousot. Tous les auteurs des chapitres du livre, spécialistes de logiciel mondialement reconnus, appartiennent naturellement à cette liste : Gérard Berry, Jean-Pierre Briot, Giuseppe Castagna, Thierry Coquand, Joëlle Coutaz, Patrick Cousot, Pierre-Louis Curien, Marie-Claude Gaudel, Jean-Marc Jézéquel, Jean-Pierre Jouannaud, Jean-Jacques Lévy, Bertrand Meyer, Michel Raynal.

Ma préface commence par l'affirmation (paradoxale au vu du titre de l'ouvrage) qu'il n'existe pas une « école française d'informatique » au sens d'un bâtiment physique, ni même au sens où l'on parle de l'école de Nancy pour l'art du verre, de l'école de Barbizon pour les peintres paysagistes ou de l'école de Vienne pour la philosophie. Les membres de l'école française d'informatique ont souvent des approches très différentes, avec par exemple parmi les auteurs mêmes du livre des différences d'appréciation entre les adeptes de la programmation par objet et de la programmation fonctionnelle, ou entre les partisans du développement en continu (« *seamless* ») et ceux de l'ingénierie des modèles. Ces amicales différence de vues sont normales, et typiques d'une communauté scientifique active, bouillonnante même. Elles ne contredisent en aucune façon la prise de conscience (la base même du livre) de ce qu'il existe indéniablement un point commun à tous ces travaux et à tous les auteurs, que je définirais comme un goût de la simplicité et de l'élégance, trouvant ses racines dans la grande tradition mathématique française (en recherche comme en enseignement). C'est cette prise de conscience qui m'a donné l'idée de construire cet ouvrage.

L'entreprise a été complexe et marquée par plusieurs péripéties, dont une assez amusante. Certains chapitres se citent entre eux et nous avions soigneusement veillé à ce que chaque référence renvoyât au bon numéro de chapitre. Mais l'un des auteurs, Patrick Cousot, cite également des chapitres de sa propre thèse. Un correcteur a estimé préférable d'inclure les titres des chapitres plutôt que leurs numéros, si bien que, dans certaines références, il a remplacé les références aux chapitres de la thèse de Cousot par des références aux chapitres de l'ouvrage lui-même… Heureusement cette série d'erreurs a été détectée et corrigée à temps.

Le quotidien d'un programmeur est d'être confronté à des éléments qui ne fonctionnent pas, sans qu'il sache pourquoi. C'est pourquoi je me souviens encore de mon émerveillement, à l'époque où j'étais étudiant, le jour où, pour la première fois, le programme de jeu de dames que j'avais écrit m'a battu. Je ressens encore la même émotion chaque fois que « ça marche » et j'espère, à travers cet exposé, vous avoir fait partager un peu de cet enthousiasme.

Présentation de *The French School of Programming*

Gérard Berry

L'ouvrage *The French School of Programming* commence par quelques rappels. La *syntaxe* désigne les règles formelles d'écriture des programmes. Les trois grands modèles de calcul sont le modèle impératif (Turing, 1936), qui prévoit l'exécution des ordres un à un ; le modèle Lambda-calcul (Church, 1936), qui repose sur un calcul fonctionnel simple mais profond ; le modèle algébrique (Kleene, autour de 1950), qui consiste à résoudre des équations en fonction. Enfin, la *sémantique* est la définition, de préférence mathématique, du sens de chaque programme écrit dans un modèle de calcul et un langage donné.

Je vais maintenant résumer chacun des chapitres de l'ouvrage, qui sont rassemblés en quatre grandes parties.

Le génie logiciel

Le test est fondamental pour que les programmes fonctionnent comme souhaité. Il est généralement considéré comme une activité purement technique et fastidieuse, et donc souvent négligé, d'autant que la plupart des programmeurs n'ont pas de formation dans ce domaine. Marie-Claude Gaudel montre comment rendre le test formel en s'appuyant sur un modèle algébrique.

Le calcul distribué est la base des réseaux et du *cloud computing*. Ses protocoles sont souvent courts mais particulièrement délicats à concevoir, et les erreurs, dangereuses, sont difficiles à corriger. Michel Raynal présente les belles améliorations qu'il a mises en œuvre dans ce domaine.

La conception et la manipulation de grands programmes exigent des techniques bien différentes de celles que nécessite leur écriture détaillée. Jean-Marc Jézéquel montre que certaines idées simples en théorie ne fonctionnent pas en pratique, sauf dans des cas spécifiques.

L'interaction homme-machine est un sujet souvent trop négligé, car il se situe aux limites de l'informatique, du design et de la psychologie. C'est pourtant une question essentielle pour les objets informatisés : voitures, avions, médical, etc. Joëlle Coutaz montre comment rendre cette approche systématique.

À cause de l'évolution de la taille et des fonctions des programme, les éléments de base de la programmation sont passés de notions simples de variables et de fonctions à des notions de comportements distribués d'acteurs communicants, ce qui exige une architecture de bien plus haut niveau, comme l'explique Jean-Pierre Briot.

Mécanismes du langage de programmation et systèmes de types

La liaison entre le monde de la syntaxe et celui de la sémantique peut-elle être rendue exacte ? Mon travail sur les fonctions stables a représenté un jalon important pour les langages fondés sur le lambda-calcul, et celui de Pierre-Louis Curien a été déterminant pour la caractérisation de la séquentialité des langages classiques et la construction de leurs modèles séquentiels.

En 1985, Thierry Coquand et Gérard Huet ont créé la Théorie des constructions, qui a révolutionné le domaine en permettant la preuve formelle en machine de théorèmes mathématiques profonds, mais aussi de programmes bien réels comme CompCert de Xavier Leroy et son équipe, le seul compilateur C certifié mathématiquement correct grâce à l'assistant de preuve Coq.

La théorie

L'interprétation abstraite, conçue en 1979 par Patrick et Radhia Cousot, s'applique à la preuve de propriétés critiques de programmes classiques grâce à des assertions démontrées vraies ou fausses par des algorithmes bien choisis. Son développement pratique, toujours étroitement relié à la théorie et conduit sur plusieurs dizaines d'années avec une équipe d'excellents chercheurs, s'est traduit par des systèmes industriels de vérification utilisés dans de nombreux domaines tels que l'automobile, l'avionique et d'autres industries critiques.

Selon le théorème de Church-Rosser, théorème fondamental du lambda-calcul de Church (1936), on peut toujours faire reconverger deux calculs différents, ce qui implique l'unicité du résultat s'il existe. Jean-Jacques Lévy va beaucoup plus loin en montrant qu'il est possible, en théorie, de réduire par paquets des expressions similaires afin d'obtenir une réduction unique de coût minimal. C'est l'un des résultats majeurs de la très riche théorie du lambda-calcul, qui permet de simplifier beaucoup de preuves de théorèmes précédents.

L'approche équationnelle soulève le problème de la terminaison (c'est à dire de l'absence de bouclage du calcul) et celui de la confluence (l'unicité du résultat des calculs terminés). Jean-Pierre Jouannaud étudie les contraintes, souvent subtiles, à imposer aux systèmes d'équations pour garantir leur terminaison et leur confluence, et prouve mathématiquement ces résultats.

Conception des langages et méthodologie de programmation

On serait fondé à croire que la programmation par objets a conquis le monde entier. En réalité, la version qui prédomine (en Java, C#, C++, Python, UML…) comporte de graves erreurs conceptuelles qui limitent les bénéfices de génie logiciel qu'on serait en droit d'en attendre : réutilisabilité, extensibilité, fiabilité. Bertrand Meyer analyse un ensemble de choix en matière de conception de langages, appliqués à Eiffel, qui montrent les bénéfices considérables d'une programmation par objets bien comprise.

Les anciens langages, comme Fortran ou Lisp, présentaient l'inconvénient de « mélanger des torchons et des serviettes », ce qui provoquait des erreurs à l'exécution. Les langages typés permettent de détecter ces erreurs dès la compilation, mais ils sont à géométrie variable, ce qui laisse plus ou moins de liberté à l'utilisateur. À travers des exemples allant du typage le plus simple au plus permissif, Giuseppe Castagna montre la variété des systèmes possibles, introduit leurs théories, mais montre aussi les difficultés liées à l'écriture des compilateurs et le coût pouvant devenir exorbitant des algorithmes de typage.

Une histoire personnelle

Dans les années 1970, l'informatique était déjà répandue dans les universités françaises, mais très peu dans les grandes écoles. Au corps des Mines, on me répétait que « L'informatique est une mode, ça va passer. » Les chercheurs français en informatique, n'ayant pas accès aux ordinateurs sur lesquels ils auraient aimé travailler, se sont fortement engagés dans la recherche théorique, ce qui leur a été très utile quand, plus tard, ils ont eu droit à ces ordinateurs. Après 1985, la pratique s'est développée et s'est bien mariée avec la théorie.

Dans les années 1980, trois équipes françaises combinant des informaticiens et des automaticiens ont inventé les langages réactifs synchrones, qui connaissent le temps et réagissent par instants successifs : Esterel, Lustre et Signal. Des problèmes sémantiques entièrement nouveaux sont apparus, mais les trois langages ont vite été adoptés par des industriels du « temps réel » (Dassault Aviation, Airbus, Safran, centrales nucléaires …), puis industrialisés. En 1990, je me suis rendu compte qu'Esterel pouvait servir aussi bien pour le hardware que pour le software, et j'ai engagé des collaborations intensives avec Cadence, Synopsys, puis Intel, STMicroelectronics et Texas Instruments.

En 2001, j'ai rejoint Esterel Technologies, entreprise à laquelle je me suis consacré à plein temps et qui a eu de nombreux clients américains et européens en hardware. En 2002, nous avons racheté Lustre/Scade pour le software et commencé à unifier Lustre et Esterel. À la fin de 2008, la crise des télécoms et l'émergence de l'iPhone a conduit à la disparition de la partie hardware d'Esterel. La partie software a été rachetée en 2012 par Ansys, qui compte plus de 300 clients en avionique et dans diverses industries. Début 2025, Ansys a été rachetée par Synopsys pour 35 milliards de dollars – preuve que la théorie française, lorsqu'elle passe à la pratique, a du bon (et que la France ne sait pas garder ses pépites) !

Le test peut aussi être formel (trente ans après)

Marie-Claude Gaudel

Il y a trente ans, l'idée que le test d'un système informatique pouvait être formel n'allait pas de soi et les tests s'opéraient de façon très pragmatique. Or, lorsqu'une erreur porte sur un seul bit, comme dans l'exemple proposé par Bertrand Meyer, il y a très peu de chance de tomber dessus en testant le système au hasard. Même si l'on applique une distribution uniforme sur l'ensemble des données, on trouvera des erreurs au début, puis de moins en moins car uniformément on ne fait que répéter des tests de cas nominaux.

Il existe de nombreuses méthodes pour tester un système informatique. Les méthodes dites de la « boîte noire » se basent sur la spécification de ce que doit faire le système, afin de définir les cas à tester et les données permettant de le faire. Elles permettent de vérifier que certains cas ou sous-cas n'ont pas été oubliés. Ce nom de « boîte noire » vient du fait que l'on ne « voit » pas le programme, par opposition à d'autres méthodes basées sur la couverture de certains éléments du programme (toutes les instructions, tous les chemins, etc.) qui visent à vérifier que tous les éléments d'un programme fonctionnent comme spécifié. Ces méthodes sont complémentaires.

Le point de départ de mes travaux a consisté à utiliser des spécifications formelles ou des modèles formels pour définir des jeux de tests. Il s'agit de décrire de façon abstraite, logique ou diagrammatique (ou les deux), certains aspects du système. En général, cette description est non-algorithmique. L'intérêt de telles spécifications est qu'elles permettent de raisonner, donc d'énumérer leurs conséquences ou leurs impossibilités.

Une spécification est d'une nature très différente de celle d'un système en fonctionnement. Pour faire le lien, il faut se baser sur une sémantique opérationnelle des spécifications et établir des hypothèses sur les systèmes testés. Dans un cadre formel, un test est une formule dérivée de la spécification qui sera interprétée par le système. On part de la notion de satisfaction d'une formule pour définir théoriquement un jeu de test exhaustif, souvent infini (toutes les conséquences et non-conséquences de la spécification), et de caractéristiques connues du système à tester et de son environnement pour poser des hypothèses de testabilité.

Un jeu de test doit cependant être fini. Pour sélectionner un sous-ensemble fini au sein d'un jeu de test exhaustif, on renforce les hypothèses de testabilité, par exemple des hypothèses d'uniformité sur des sous-ensembles de données, des hypothèses de régularité pour limiter la taille des données de test, des hypothèses d'atomicité, etc. Ainsi, pour tester un programme calculant la valeur absolue d'un nombre entier, on va proposer de réaliser le test pour un nombre négatif, pour un nombre positif et pour zéro, cette démarche correspondant à une hypothèse d'uniformité sur les sous-ensembles de données. Avec des hypothèses fortes, on effectuera un petit nombre de tests, et un grand nombre de tests si les hypothèses sont faibles.

On doit d'abord prouver que sous les hypothèses de testabilité, le succès du jeu de test exhaustif est équivalent à la satisfaction de la spécification, puis que le renforcement des hypothèses et la réduction du jeu de test préservent cette équivalence.

L'abstraction est essentielle pour définir les spécifications, mais elle pose des problèmes d'observabilité des résultats. La décision du succès ou de l'échec d'une expérience de test demande, dans certains cas, une infinité d'observations, d'où la nécessité d'introduire des hypothèses d'observabilité en plus des hypothèses de testabilité. On peut les vérifier par analyse statique du programme, ou en les prouvant (par exemple à l'aide du théorème prouveur HOL/TestGen), ou par test probabiliste (par exemple, un tirage uniforme intensif sur un sous-domaine d'uniformité). S'il s'avère impossible ou trop coûteux de les vérifier, ce qui est fréquemment le cas, on doit s'accommoder de la situation mais, du moins, a-t-on exprimé ces hypothèses qui seront alors fournies avec le résultat des tests.

La modélisation en logiciel, des outils CASE à l'apprentissage machine

Jean-Marc Jézéquel

La complexité de l'informatique présente trois dimensions largement orthogonales. La première est intrinsèque et liée à l'indécidabilité du moindre programme informatique, déjà évoquée. La deuxième dimension de la complexité n'est pas propre à l'informatique, mais mais consubstantielle de la taille de l'objet produit, dans quelque domaine que ce soit. Par exemple, pour construire l'ensemble de bâtiments dans lequel nous nous trouvons, il n'y a pas d'autre solution que de scinder le projet en différentes parties, de distribuer le travail puis d'assembler le résultat. Le concept de modularité en génie civil ayant été inventé par Gustave Eiffel, j'imagine que ce n'est pas par hasard que Bertrand Meyer a donné ce nom d'Eiffel au langage qu'il a conçu afin d'y appliquer cette idée La troisième dimension de la complexité de l'informatique, moins souvent évoquée, est la variabilité des programmes informatiques, dans laquelle on peut distinguer la variabilité venant du bas et celle venant du haut.

La variabilité venant du bas

Un algorithme est un objet mathématique pur mais, dès qu'il commence à être exécuté par une machine, il est confronté au monde physique et perd cette pureté. Par exemple, pour des raisons d'efficacité, un processeur moderne n'exécute pas toujours ses instructions dans le bon ordre, constate parfois qu'il s'est trompé, revient en arrière et essaie à nouveau, ce qui, additionné à une gestion complexe des caches mémoire, rend le temps d'exécution ou la consommation d'énergie non déterministes. Autre exemple, si un sous-circuit s'avère défaillant, le processeur s'adapte dynamiquement pour compenser cette défaillance, en sorte que, si vous testez un programme à plusieurs reprises au cours du temps avec la même requête dans des conditions exactement similaires, vous pourrez obtenir quand même un temps de réponse différent et une consommation d'énergie différente. Le résultat, en revanche, restera le même, sauf lorsque l'algorithme prend en compte la dimension du temps.

C'est le cas, par exemple, pour les algorithmes de reconnaissance permettant aux radars de distinguer un missile d'un oiseau. L'objectif est d'obtenir une réponse approximative au bout d'une seconde, plutôt que d'attendre dix secondes pour obtenir une réponse parfaite. L'algorithme étant borné par le temps, on risque d'obtenir des réponses différentes malgré des données d'entrée identiques. Le non-déterminisme des résultats peut également être lié au fait que les machines fonctionnent en réseau et que le temps de propagation des données à travers le monde peut varier.

La variabilité venant du haut

À ceci s'ajoute la variabilité venant du haut, déjà mentionnée par Bertrand Meyer, liée au fait que le commanditaire peut, en cours de route, changer d'avis sur les spécifications du programme, ou que, à l'échelle de la planète, les clients potentiels vivent sous des lois différentes et peuvent avoir des usages variés du logiciel. Cette variabilité est devenue centrale dans la façon dont on conçoit les logiciels. L'un des systèmes les plus utilisés, Linux, comprend 15 000 options de configuration. Si ces options sont binaires, cela représente $2^{15\,000}$ possibilités, ou $10^{5\,000}$, à comparer avec le nombre d'atomes présents dans l'univers, estimé à 10^{100}.

La seule façon de gérer cette complexité consiste à modéliser chacun des facteurs de variation en les décrivant d'un point de vue donné (la sécurité, les données, la dynamique du système…), toute la difficulté étant, ensuite, de tout assembler. À l'heure actuelle, malgré de nombreuses tentatives au cours de l'histoire, il n'existe pas d'opérateur de composition fonctionnant de manière satisfaisante. Dans mon chapitre, je passe en revue un ensemble de bonnes idées qui, à telle ou telle époque, ont paru convaincantes au point de conduire à tenter de les mettre en œuvre de façon industrielle, puis ont dû être abandonnées ou au mieux cantonnées à certaines classes de problèmes.

Avec l'apparition de l'apprentissage machine, cette variabilité est encore moins maîtrisée, car ce qui sera produit demain n'est pas connu au moment de la conception du programme. Lorsque l'on construit une cathédrale, on n'imagine pas qu'un jour, on pourrait avoir besoin de remplacer ses piliers. Dans le cas d'un logiciel, cette possibilité fait partie des données du problème.

Des procédures, objets, acteurs, composants et services aux agents

Jean-Pierre Briot

Comme le soulignent Bertrand Meyer et Jean-Marc Jézéquel, un des grands enjeux du concepteur/programmeur est de tenter d'anticiper des besoins d'évolution du programme et ainsi sa variabilité, tout en garantissant la capacité de son exécution et autant que possible la vérification de propriétés (telles que la terminaison, la cohérence…). Un élément important des langages de programmation consiste à proposer des abstractions (fonction, procédure, module, objet, message, asynchrone, acteur, intention, service…) ayant un pouvoir d'expressivité et une flexibilité adéquats et offrant des possibilités de vérification suffisantes, ces deux critères étant en tension. Dans mon chapitre, je classe ces abstractions selon trois axes, les deux premiers ayant trait à cette flexibilité recherchée, et je m'efforce de montrer comment elles ont évolué au fil du temps.

La flexibilité de la sélection de l'action

Le premier axe, celui de la flexibilité de la sélection de l'action, est illustré par l'anecdote de Bertrand Meyer sur la confusion entre les chapitres de la thèse d'un des auteurs du livre et les chapitres du livre. Comment nommer sans ambiguïté les variables ou les procédures, tout en ménageant une certaine flexibilité afin de permettre, par exemple, de redéfinir une procédure ?

Au fil du temps, la liaison entre la dénomination de la procédure et le code effectif s'est effectuée de façon de plus en plus tardive. Dans la programmation « Monolitique » (de type Fortran), l'invocation est globale, numérique et statique, et elle prend la forme d'un saut (goto). Dans la programmation modulaire (de type Pascal, C++), elle est symbolique et statique, et prend la forme d'un appel de procédure. Dans la programmation par objets (Java, Eiffel), elle est dynamique et se traduit par un appel de méthode. La sélection de la méthode dépend de la classe de l'objet valeur d'une variable au moment de l'exécution et gagne ainsi en dynamicité. Enfin, dans la programmation par agents (AgentSpeak), elle est dynamique et contextuelle ; elle peut être dirigée, par exemple, par les buts. Par opposition à l'algorithme, où le programmeur prévoit et spécifie tout à l'avance, les décisions et initiatives sont alors confiées à des agents d'intelligence artificielle dotés de

connaissances, d'heuristiques, de stratégies, de capacités de planification, d'apprentissage, etc. En revanche, au fil de ces évolutions, les vérifications deviennent de plus en plus difficiles à effectuer.

La flexibilité du couplage

Pour illustrer le deuxième axe, celui de la flexibilité du couplage, prenons le cas d'un objet A dont la variable référence un objet B afin de lui transmettre des informations. Si l'on souhaite que A puisse également transmettre des informations à C, il faut ajouter une référence de A vers C. Malheureusement, une variable ne peut avoir qu'une seule valeur. La solution consiste à remplacer la valeur par une collection et à prévoir une itération. Mais ceci oblige à intervenir dans la représentation (implantation) de l'objet A et ainsi de briser le principe d'encapsulation, alors qu'il ne s'agit que de reconnecter A également à C. En s'inspirant de l'électronique, on peut considérer que les références internes sont externalisées et deviennent des objets de première classe, des « connecteurs », qui peuvent être manipulés dynamiquement, et ainsi ajouter une connexion de A à C sans toucher à son implantation. On peut même laisser aux entités logicielles la capacité d'effectuer elles-mêmes ces reconnexions. Par exemple, l'entité effectue une recherche des services qui lui conviennent pour un objectif donné et sélectionne puis « contracte » avec l'un d'entre eux. Cette fois, l'évolution des abstractions est la suivante : saut, modules, objets, acteurs, composants, services, agents.

Le niveau d'abstraction

En informatique, lorsqu'on ne parvient pas à résoudre un problème, on monte en abstraction et on recourt à la modélisation, comme Jean-Marc Jézéquel l'a évoqué. L'évolution des abstractions pour ce troisième axe est : octets, structures de données, objets/messages, services, modèles/ontologies, agents/intentions/plans. On passe ainsi progressivement des données les plus élémentaires (et proches de la machine) à des connaissances. Un niveau d'abstraction plus élevé permet une expression plus proche des concepts et processus identifiés lors de la conception d'une application, plus de portabilité également, et aussi possiblement des capacités plus abstraites d'analyse et de vérification. Mais, de manière duale, cela nécessite également des mécanismes plus sophistiqués pour la traduction d'un niveau plus abstrait et éloigné de la machine (via par exemple une conception via une ingénierie des modèles) jusqu'à une implantation effective et optimisée. Sans oublier bien évidemment des garanties de correction de la traduction (préservation du comportement du programme).

En résumé

Dans une approche conservatrice, pertinente pour des systèmes critiques, on mobilisera un typage statique fort, des spécifications formelles et des garanties, mais ce dispositif bridera la flexibilité et la dynamicité. Dans une approche dynamique/optimiste, on recourra à un typage dynamique, avec reconfigurabilité, délégation de décision aux entités pour tenter de résoudre les problèmes et moins de garanties. On utilisera alors, par exemple, une approche de programmation par contrats ou/et de tests, plutôt qu'une vérification statique de programmes, en tentant d'identifier un jeu de tests avec une bonne couverture des nombreux cas possibles, tout en restant fini. L'approche proposée par Marie-Claude Gaudel dans son chapitre vise justement à optimiser à la fois la taille et la couverture de jeux de tests, automatiquement générés à partir de spécifications formelles. Ceci illustre ainsi une forme de chemin du milieu et de complémentarité des deux approches (spécifications formelles/typage fort ou typage/adaptation dynamique). En effet, aucune des deux n'est dans l'absolu meilleure que l'autre : tout dépend des besoins ou des exigences auxquels on veut répondre. On peut ainsi tenter de fixer le curseur au niveau adéquat suivant les besoins de l'application envisagée. Un exemple est de rajouter un typage en partie statique, ou au contraire flexibiliser une gestion de ressources en suivant une approche d'implémentation ouverte (dont certains aspects sont réifiés et ainsi manipulables au niveau du langage).

Mots-clés : Génie logiciel, indécidabilité, langage de programmation, programmeur, test de système informatique, variabilité des programmes informatique.

Citation : Bertrand Meyer, Gérard Berry, Marie-Claude Gaudel, Jean-Marc Jézéquel & Jean-Pierre Briot. (2025). *La programmation : industrie des idées pures*. Les séances thématiques de l'Académie des technologies. @

Retrouvez les autres parutions de l'Académie des technologies sur notre site academie-technologies.fr
Académie des technologies. Le Ponant, 19 rue Leblanc, 75015 Paris. 01 53 85 44 44

Production du comité des travaux.
Directeur de la publication : Patrick Pélata
Rédacteur en chef de la série : Béatrice Lathuile
Auteur : Élisabeth Bourguinat
n° ISSN : 2826-6196