# Debuggage, vérification, certification de code

Tous les utilisateurs de l’informatique ont affaire, à un moment ou l’autre, à des *bugs* : des dysfonctionnements plus ou moins graves, souvent agaçants, mais qui parfois ont des conséquences importantes (vol de données, interception de communications, pertes d’engins spatiaux…). Les *bugs* sont-ils une fatalité ? Ils ont fait l’objet, depuis les débuts de l’informatique, d’études tant théoriques qu’appliquées ; le problème est, comme on le verra, très difficile !

## La difficulté à spécifier ses attentes

Tout d’abord, considérons la définition. Un bug, c’est, dira-t-on, quand l’ordinateur fait quelque chose d’incorrect. Mais qu’est-ce que l’incorrect ? Si l’on s’accordera à considérer qu’une machine qui se fige, un traitement de textes qui affiche un message sibyllin avant de se fermer en perdant le document en cours de modifications, un écran bleu rempli de signes cabalistiques, sont des évènements indésirables, il n’est parfois pas évident de définir ce qui correct ou incorrect !

Le logiciel, en effet, est censé répondre à des besoins humains pas forcément bien définis, et même parfois contradictoires. Ainsi, dans une version préliminaire des textes réglementaires gouvernant ParcourSup, ce système devait accorder une priorité aux boursiers, et une priorité aux bacheliers locaux ; mais le texte ne précisait pas comment départager ces deux priorités.[[1]](#footnote-2) Lorsque des règles imprécises ou contradictoires sont appliquées par des humains, ceux-ci peuvent se rapporter à leur jugement personnel, le recours à leur hiérarchie, voire un contentieux devant les tribunaux ; mais la machine a besoin de précision. Les exigences imprécises ou incohérentes ne sont pas l’apanage des textes législatifs et réglementaires ; les ingénieurs ont également du mal à définir ce qu’ils attendent d’un système. Des exigences trop laxistes peuvent laisser passer des comportements en fait indésirables («vous n’aviez jamais dit qu’il fallait que le logiciel interdise de réserver un train à une date déjà passée !»), des exigences trop strictes peuvent être impossibles à réaliser. Il existe même des logiciels destinés à chercher des bugs dans leurs spécifications ![[2]](#footnote-3)

Une fois que l’on sait *ce que l’on veut faire faire* à l’ordinateur, il faut concevoir *comment* il doit le faire. Ce n’est évidemment pas la même chose que de savoir qu’on veut, par exemple, trier des données, et de détailler comment la machine doit s’y prendre pour parvenir à ce résultat. De fait, il existe quantité de procédés de calcul, ou *algorithmes*, qui permettent de trier des données, qui ont des performances différentes et sont plus ou moins adaptés à telle ou telle situation (les données tiennent-elles toutes dans la mémoire vive de la machine, ou sont elles dans une mémoire externe plus lente, etc.). Ceci demande une analyse du problème. Ensuite, il faut programmer, c’est-à-dire traduire le fruit de cette analyse en un *programme*, c’est-à-dire des instructions qu’un ordinateur puisse exécuter. Là encore, ces activités peuvent donner lieu à erreur, et donc à des *bugs*.

Le cas d’un logiciel qui ne fonctionnerait jamais serait simple à régler : on s’en rendrait compte immédiatement. En réalité, on a le plus souvent affaire à des logiciels qui ont des comportements incorrects dans certaines circonstances : quand quelque chose se produit à un moment bien précis, quand on l’utilise en conjonction avec telle nouvelle ou ancienne version d’un autre logiciel, plus généralement quand telle éventualité qui n’avait pas été prise en compte au moment de l’analyse ou de la programmation s’avère en fait possible. Isoler les circonstances où le bug survient est souvent difficile, tant il y a de facteurs ; le risque étant que les développeurs du logiciels répondent « chez moi, ça marche » (ou, plus poliment « nous n’arrivons pas à reproduire ce comportement »).

Les bugs peuvent poser des problèmes de *sécurité informatique*: un dysfonctionnement peut être exploité avec adresse afin de permettre son exploitation à des fins d’intrusion, vol de données, exigences de rançon… Les individus malintentionnés peuvent pour cela créer des circonstances très spécifiques et improbables en usage normal.

## Une approche classique : le test

Pour éviter que les bugs ne surviennent « en production », c’est-à-dire une fois le système informatique en fonctionnement, une approche classique est de le tester. On fera des *tests unitaires* (vérifiant que de petites portions de logiciels font bien ce qu’on attend d’elles), des *tests d’intégration* (sur des portions plus vastes)… On pourrait utiliser des outils d’*intégration continue*, avec des batteries de tests lancées automatiquement en cas de modification du logiciel. Le test souffre toutefois de deux difficultés : la rédaction des cas de tests représentatifs est fastidieuse et ingrate pour les développeurs, de sorte qu’ils ont souvent tendance à s’y consacrer a minima ; et le test ne dit rien sur les éventualités non testées. Pour le premier point, on peut éventuellement utiliser des outils de génération de tests, plus ou moins « intelligents », ou encore des outils de mesure de « couverture » de tests (est-ce qu’au moins on est passé dans tous les morceaux du programme, etc.). Le second point est plus délicat : même si tous les tests passent, cela ne démontre pas qu’il n’existe aucun autre cas où il y aurait dysfonctionnement.

En mathématique, pour *démontrer* qu’un théorème est vrai dans tous les cas, on produit une démonstration. Un ordinateur exécutant des calculs bien définis mathématiquement (des additions, des mouvements de données…), on a très tôt pensé à démontrer mathématiquement que des programmes se comportent correctement (A. Turing, *Checking a large routine*, 1949), puis on a développé des formalismes mathématiques spécifiquement adaptés (logique de Hoare, etc.). Ces démonstrations étant particulièrement fastidieuses, on a songé d’une part à les automatiser totalement ou partiellement (*model-checking*, *software model-checking*, analyse statique par interprétation abstraite, etc.), d’autre part à pallier les erreurs humaines de raisonnement en faisant relire et valider la démonstration mathématique par une machine (*preuve assistée par ordinateur[[3]](#footnote-4)*).

On regroupe toutes ces applications du raisonnement mathématique au bon fonctionnement des systèmes informatiques sous le nom de *méthodes formelles*. Celles-ci étaient toutefois peu développées en dehors d’usages à très haut niveau de sûreté (commandes de vol informatisées d’avions,[[4]](#footnote-5) systèmes de pilotage de métros, de trains et d’aiguillages…), en raison de leur coût et du haut niveau de technicité des recrutements nécessaires. Des méthodes semi-formelles, plus automatisées et à moindre coût d’entrée pour les utilisateurs ont été développées, notamment pour répondre aux besoins des grands acteurs du domaine.[[5]](#footnote-6)

Dans les domaines à haut niveau de sûreté, on se méfie de tout. Ainsi, même si le programme tel que rédigé par les développeurs est correct, il se peut que sa traduction dans le langage machine, effectuée par un logiciel spécialisé appelé *compilateur*, soit incorrecte. Dans certains domaines comme l’avionique, ceci obligé à de fastidieuses et coûteuses vérifications. Un grand succès des méthodes formelles est CompCert, un compilateur muni d’une preuve mathématique de correction.[[6]](#footnote-7)

## Des problèmes en lien avec de grandes questions mathématiques

Pourrait-on automatiser entièrement la recherche de bug, ou la démonstration mathématique de leur absence ? C’est une question bien plus profonde qu’il n’y paraît, car elle touche à des problèmes fondamentaux en mathématiques. Un résultat fondamental de Turing est qu’il n’existe pas d’algorithme qui, au vu d’un programme, puisse répondre à une question aussi simple que « ce programme va-t-il finir par produire un résultat, ou va-t-il au contraire continuer à calculer pendant un temps infini ». Ce résultat a été démontré, il faut le souligner, sur un modèle théorique de ce que pourrait être le calcul automatisé, alors qu’on n’avait pas encore d’ordinateurs ! Le théorème de Rice généralise ce résultat : quelle que soit la spécification que l’on considère, il n’existe pas de façon automatique et fiable de déterminer si un programme la respecte. Ces résultats de *théorie de la calculabilité* sont liés à des considérations plus générales de logique mathématique (travaux de Gödel, Tarski, etc.) relatifs à la possibilité de décider de la vérité de théorèmes mathématiques, à la suite notamment de questionnements de Hilbert.

Même dans les cas restreints où des approches automatiques sont possibles,[[7]](#footnote-8) se pose la question de leur coût. La *théorie de la complexité algorithmique* tente de séparer les problèmes en classes de difficulté croissante, et, malheureusement, la plupart des problèmes intéressants en matière d’analyse de programme appartiennent à des classes difficiles. Les recherches portent d’une part sur la possibilité de séparer ces classes de difficulté (le problème dit « P vs NP » est considéré comme un des grands problèmes mathématiques du millénaire), d’autre part sur des approches algorithmiques pragmatiques, efficaces sur de nombreux problèmes pratiques, même si elles échouent dans le cas général à produire un résultat en temps acceptable.

## Et l’intelligence artificielle ?

Dans les approches classiques de l’*algorithmique*, c’est-à-dire de la conception des algorithmes, on part d’une définition mathématiquement précise de ce que l’on veut faire. Parfois, il y a des subtilités : « classer une liste d’étudiants par ordre alphabétique de noms de famille » ne précise pas comment on classe entre eux les étudiants ayant le même nom de famille, et ne précise pas non plus la notion d’ordre alphabétique en présence d’espaces, de caractères accentués et autres signes diacritiques (faut-il classer Dupre avant ou après Du Pré ou Dupré, etc.). Toutefois, une fois ces questions réglées, le problème est bien défini. Or, nous désirons souvent résoudre des problèmes mal définis. On a ainsi voulu appliquer « la méthode des algorithmes » à la recherche de potentiels terroristes ; mais on peinerait à vouloir définir mathématiquement cette notion de potentiel terroriste. De même, même si nous reconnaissons facilement un chat d’un chien, nous serions bien en peine de fournir une définition mathématique permettant de distinguer leurs photographies. Pour ce genre de questions, on a de plus en plus recours à des techniques d’intelligence artificielle, notamment issues de l’*apprentissage profond*, qui souvent tentent de produire une règle générale à partir d’exemples pour lesquels les humains ont donné la bonne réponse. C’est ainsi que quand, pour accéder à un service en ligne, nous répondons à des *captchas* censés prouver que nous sommes des humains et non des robots, nous entraînons des intelligences artificielles à savoir reconnaître des feux de circulation, des motocyclettes, ou d’autres objets. La question de la correction de ces systèmes porte ainsi sur la couverture de la base d’entraînement (est-elle représentative ?), sur les critères retenus par le processus d’apprentissage et sur leur robustesse (est-ce qu’un changement de détail minime peut produire une décision incorrecte ?).

## Glossaire

**Algorithme** Description des étapes d’un procédé automatique de calcul. Par exemple, l’algorithme de *tri par sélection* peut se décrire ainsi : rechercher le plus petit élément dans la table, l’intervertir avec l’élément en première position, et procéder de même pour le reste de la table. Ce terme a récemment acquis une connotation d’opacité, alors que dans son acception classique il désigne un procédé bien décrit et susceptible d’analyse mathématique.

**Déboguage** La recherche et l’élimination des bugs, un travail très fastidieux, parfois très subtil intellectuellement, et qui se prolonge même une fois le logiciel diffusé. Des outils spécifiques (*debuggers*, instrumenteurs de code, *fuzzers*, environnements d’exécution spéciaux…) peuvent aider.

**Logiciel ou programme** Instructions qu’un ordinateur va suivre pour mettre en œuvre des fonctionnalités, rédigées dans un **langage de programmation**. On le distingue de l’algorithme, forme plus idéalisée de certaines fonctionnalités.

**Spécification** Définition de ce qu’un logiciel est censé faire. Une spécification peut être en langue naturelle (français, anglais…), entièrement mathématisée, ou à mi-chemin entre les deux. On dit qu’un programme est **correct** s’il répond à sa spécification.

**Sûreté de fonctionnement** Propriété d’un système informatisé de ne jamais produire un résultat incorrect. C’est une question liée à la **sécurité**, où il faut se prémunir d’attaques intentionnelles. Par exemple, un traitement de textes peut être sûr s’il relit correctement les fichiers qu’il produit, mais souffrir de failles de sécurité exploitables en lui fournissant des fichiers subtilement altérés et provoquant chez lui des comportements indésirables.

## Bibliographie

On pourra se rapporter aux cours de Xavier Leroy au Collège de France.

1. Cette imprécision de rédaction a été soulevée par des algorithmiciens que l’on avait sollicités sur ce texte. [↑](#footnote-ref-2)
2. Par exemple, le logiciel *Stimulus*, de la jeune pousse *Argosim*, incubée au laboratoire Verimag, depuis rachetée par Dassault Systèmes. [↑](#footnote-ref-3)
3. Citons notamment l’outil Coq, développé en France. [↑](#footnote-ref-4)
4. Par exemple l’outil Astrée, développé au laboratoire d’informatique de l’École normale supérieure (CNRS / ENS), est utilisé notamment par l’avionneur Airbus et par l’équipementier automobile Bosch. [↑](#footnote-ref-5)
5. Citons par exemple l’analyseur *Infer*, de Facebook/Meta, ou encore le projet SAGE de Microsoft. Les importantes marges dégagées internationalement par ces entreprises leur permettent de se doter d’équipes internes de recherche. [↑](#footnote-ref-6)
6. Citons parmi les principaux concepteurs de CompCert Xavier Leroy, devenu professeur de sciences du logiciel au Collège de France, et Sandrine Blazy, médaille d’argent 2023 du CNRS. [↑](#footnote-ref-7)
7. On se place dans ces cas en restreignant notamment l’espace mémoire ou le temps d’exécution des programmes analysés. Les résultats d’indécidabilité de Turing et de Rice ne s’appliquent alors plus. [↑](#footnote-ref-8)