Déboguage, vérification et certification de code

David Monniaux

Déboguage, vérification et certification de code

David Monniaux

Tous les utilisateurs d'outils informatiques ont déjà eu affaire à des « bugs » : des dysfonctionnements plus ou moins graves, souvent agaçants, parfois avec des conséquences importantes (vol de données, interception de communications, pertes d'engins spatiaux…). Nous avons pris l'habitude d'en parler comme d'une fatalité, mais ils sont liés à de profondes questions scientifiques.

La difficulté à spécifier ses attentes

En général, un utilisateur range dans la catégorie des « bugs » tout ce que son appareil informatique fait d'incorrect. S'il est bien sûr indésirable qu'une machine se fige, qu'un traitement de textes affiche un message sibyllin et se ferme en perdant le document en cours, ou que Windows produise un écran bleu rempli de signes cabalistiques, il n'est parfois pas évident de définir ce qui est correct ou incorrect.

Le logiciel doit répondre à des besoins humains souvent mal définis et parfois contradictoires. Ainsi, dans une version préliminaire des textes gouvernant ParcourSup (plateforme d'admission aux études post-Bac), ce système devait accorder la priorité aux boursiers et aux bacheliers locaux ; mais on ne précisait pas comment départager ces deux cas. Lorsque des règles imprécises ou contradictoires sont appliquées par des humains, ceux-ci peuvent se reporter à leur jugement personnel, à leur hiérarchie, voire aux tribunaux ; mais la machine a besoin de précision. Des exigences trop laxistes peuvent laisser passer des comportements indésirables (« Vous n'aviez jamais dit qu'il fallait que le logiciel interdise de réserver un train à une date déjà passée ! »), tandis que des exigences trop strictes peuvent être impossibles à réaliser. Il existe même des logiciels, comme *Stimulus*, destinés à chercher des bugs dans des spécifications, donc bien en amont du code.

Une fois que le donneur d'ordres a déterminé ce qu'il voulait *faire faire* à l'ordinateur (par exemple, trier des données), le concepteur du logiciel doit réfléchir à *comment* le faire. Ainsi, il existe quantité de procédés de calcul, ou « algorithmes »*, pour trier des données, mais ils ont des performances différentes et sont plus ou moins adaptés à chaque situation (quantités de données, mémoire disponible…). Il faut donc réaliser une *analyse* du problème, puis traduire le fruit de cette analyse en un *programme*, c'est-à-dire des instructions qu'un ordinateur peut exécuter. Là encore, ces activités peuvent donner lieu à des erreurs, et donc à des bugs. La première ligne de défense consiste à mettre en œuvre une conception claire, une programmation rigoureuse, une relecture sérieuse et des bonnes pratiques. Mais souvent, cela ne suffit pas.

Un logiciel* qui ne fonctionne jamais est un cas facile, car l'utilisateur s'en aperçoit immédiatement. Mais, dans la majorité des cas, le bug survient seulement dans certaines circonstances : quand quelque chose se produit à un moment bien précis, quand le logiciel est utilisé avec telle nouvelle ou ancienne version d'un autre logiciel, plus généralement quand telle éventualité non prise en compte au moment de l'analyse ou

de la programmation s'avère en fait possible. Isoler les circonstances dans desquelles le bug survient est souvent difficile, tant il y a de facteurs ; le risque étant que les développeurs du logiciel répondent « chez moi, ça marche ».

Situation plus préoccupante : les bugs peuvent poser des problèmes de sécurité* informatique. En effet, une faille dans un logiciel peut être exploitée avec adresse à des fins d'intrusion, de vol de données, ou d'exigences de rançon… Les individus malintentionnés peuvent pour cela exploiter des failles 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*. Le développeur réalise des *tests unitaires* (il vérifie que de petites portions de logiciels font bien ce qu'il attend d'elles) et des *tests d'intégration* (sur des portions plus vastes), vérifiant qu'un nombre d'unités logicielles dépendantes fonctionnent bien ensemble. Il peut aussi utiliser des outils d'*intégration continue*, avec des batteries de tests lancés automatiquement en cas de modification du logiciel.

Bien tester est difficile. Rédiger des cas de tests représentatifs est fastidieux et ingrat pour les développeurs, de sorte qu'ils ont souvent tendance à s'y consacrer *a minima*. Des outils de génération de tests, plus ou moins « intelligents », ou encore des outils de mesure de *couverture* de tests (pour vérifier que tous les morceaux du programme ont été couverts), peuvent aider. Plus important : même si tous les tests passent, cela ne démontre pas qu'il n'existe aucun autre cas où il y aurait dysfonctionnement.

Pour démontrer qu'un théorème est vrai dans tous les cas, un mathématicien produit une *démonstration*. L'ordinateur, lui, exécute des calculs bien définis mathématiquement (des additions, des mouvements de données…). Des informaticiens tels qu'Alan Turing* ont donc très tôt pensé à démontrer mathématiquement que des programmes se comportaient correctement, puis ils ont développé des formalismes mathématiques spécifiquement adaptés (logique de Hoare*, etc.). Ces démonstrations étant très fastidieuses, ils ont songé d'une part à les automatiser totalement ou partiellement (*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, avec des outils tels que Coq ou Lean).

Les *méthodes formelles* sont des applications du raisonnement mathématique au bon fonctionnement des systèmes informatiques. Celles-ci sont encore peu développées en dehors d'usages à très haut niveau de sûreté* (systèmes de pilotage de métros, de trains et d'aiguillages, centrale nucléaires…), en raison de leur coût, de la haute qualification des recrutements nécessaires et de problèmes de passage à l'échelle, mais se répandent. Ainsi l'outil Astrée, développé à l'École normale supérieure, a été créé pour des applications à haut niveau de sûreté (commandes de vol informatisées d'avions), mais a ensuite trouvé des clients dans des industries moins exigeantes (automobile…). Des méthodes semi-formelles, moins rigoureuses, plus

automatisées et à moindre coût d'entrée pour les utilisateurs ont été développées, notamment pour répondre aux besoins des grands acteurs du domaine (analyseur *Infer*, de Facebook/Meta, ou projet SAGE de Microsoft).

Dans les domaines à haut niveau de sûreté, comme l'avionique, il est crucial de se méfier de tout et les ingénieurs procèdent donc à des vérifications fastidieuses et coûteuses. Même si le programme tel que rédigé par les développeurs est correct, sa traduction dans le langage machine, effectuée par un logiciel spécialisé appelé « compilateur », peut être incorrecte. *CompCert* – développé par Xavier Leroy, professeur au Collège de France, et Sandrine Blazy, médaille d'argent 2023 du CNRS – est le seul compilateur utilisable industriellement muni d'une preuve mathématique de correction.

Des problèmes en lien avec de grandes questions mathématiques

Pourrait-on automatiser entièrement la recherche de bugs, ou la démonstration mathématique de leur absence ? Cette question est bien plus profonde qu'il n'y paraît, car elle touche à des problèmes fondamentaux en mathématiques, notamment ce résultat de Turing : il n'existe pas d'algorithme qui puisse répondre à une question aussi simple que « ce programme va-t-il finir par produire un résultat ou va-t-il continuer à calculer pendant un temps infini ? ». Il faut cependant rappeler que ce résultat a été démontré sur un modèle théorique de ce que pourrait être le calcul automatisé, à une époque où les ordinateurs n'existaient pas ! Le « théorème de Rice » généralise ce résultat : *quelle que soit la spécification considérée, 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, relatives à la possibilité de décider de la vérité de théorèmes mathématiques.

Même dans les cas où des approches automatiques sont possibles (espace mémoire ou temps d'exécution des programmes restreints), 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 (la conjecture dite « P vs NP » est considérée comme un des grands problèmes mathématiques du millénaire), et 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 l'approche classique de l'algorithmique, c'est-à-dire de la conception et de l'analyse des algorithmes, le développeur part d'une définition mathématiquement précise de ce qu'il veut faire. Parfois, il y a des subtilités : ainsi « classer une liste par ordre alphabétique » ne précise pas cette notion en présence d'espaces, de caractères accentués (faut-il classer Dupre avant ou après Du Pré ou Dupré ?). Toutefois, ces questions réglées, le problème est bien défini. Or, nous désirons souvent résoudre des problèmes mal définis. Certains ont voulu appliquer des algorithmes à la recherche de potentiels terroristes ; mais qui peut

définir mathématiquement cette notion de « potentiel terroriste » ? En outre, même si nous reconnaissons facilement un chat d'un chien, nous peinerions à fournir une définition mathématique distinguant leurs photographies.

Pour répondre à ces problématiques, les chercheurs ont souvent recours à des techniques d'intelligence artificielle, notamment issues de l'« apprentissage profond », qui tentent de produire une règle générale à partir d'exemples pour lesquels les humains ont donné la bonne réponse. Ainsi, 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 sa qualité (contient-elle beaucoup de mauvaises réponses ?), 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 ?).

Conclusion

Les problèmes informatiques ne se réduisent pas simplement à des problèmes de programmation : ils se nichent souvent dans une mauvaise compréhension du problème à résoudre, une mauvaise documentation, des interactions non maîtrisées avec d'autres logiciels ou des composants physiques, voire une interface utilisateur peu intuitive. Les méthodes issues d'approches mathématiques du logiciel ont leur rôle à jouer, mais aussi leurs limites (coût, manque de flexibilité). Le développement de langages de programmation avec moins de chausse-trappes que les langages historiques (par exemple, Rust qui commence à remplacer le langage C) permet d'éviter certaines catégories de bugs. L'intelligence artificielle apporte des solutions (assistance à la rédaction de programmes), mais pose d'autres problèmes (imitation de l'existant, y compris dans ses erreurs). Nous sommes donc loin d'avoir épuisé la question !

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 et de critique.

Apprentissage. En intelligence artificielle, l'apprentissage dirigé consiste à régler des paramètres d'un logiciel à partir d'un ensemble d'exemples : on donne pour chacun la réponse attendue. Par exemple, on donnera un ensemble de photographies chacune étiquetée avec le sujet représenté, afin que le logiciel « apprenne » à reconnaître les sujets représentés, y compris et surtout sur des photographies non présentes dans son corpus d'apprentissage. Dans l'**apprentissage profond**, les paramètres à régler sont ceux d'un réseau de neurones artificiels multicouche ; les « neurones » sont ici des petits éléments de calcul numérique inspirés (avec une certaine distance) du fonctionnement des neurones humains.

Bug. Mot anglais signifiant insecte, punaise, microbe, utilisé en informatique pour nommer un défaut d'un programme informatique qui aboutit à un dysfonctionnement. En 1947, un insecte se serait bloqué dans le calculateur Mark II de l'université Harvard aux États-Unis causant des erreurs. Il semble, cependant, que le terme ait été utilisé bien avant par des ingénieurs.

Captcha. Question posée par certains sites pour s'assurer que l'utilisateur qui y accède est bien un humain, et non un robot, par exemple pour éviter un afflux de requêtes automatisées. Un captcha peut demander de reconnaître visuellement un texte distordu ou de reconnaître des éléments (feux de circulation, bouche d'incendie, bicyclette…) sur une photographie urbaine. Ce dernier type de captcha sert fort probablement à entraîner des « intelligences artificielles » destinées à piloter des véhicules autonomes, comme auparavant des captchas où l'on demandait aux utilisateurs de reconnaître des textes imprimés scannés servaient à entraîner des systèmes de reconnaissance de caractères destinés à la numérisation de masse. Il s'agit là d'une forme de travail exigé des utilisateurs, que l'on peut soit considérer comme du travail gratuit, soit comme le prix d'accès au service.

Compilateur. Les humains écrivent les logiciels dans des langages de programmation qui leur permettent d'exprimer ce qu'ils veulent en s'abstrayant des détails de fonctionnement de la machine, avec plus ou moins d'abstraction et de lisibilité suivant le langage choisi. Ces programmes ne sont pas directement exécutables sur la machine, qui ne comprend que des instructions élémentaires très simples (opérations arithmétiques, accès dans la mémoire…) formant le *langage machine* (voir ce nom). Un compilateur est un traducteur d'un langage de programmation vers le langage machine. Un **interpréteur** va quant à lui interpréter instruction par instruction au lieu de traduire d'une traite. La distinction entre interpréteur et compilateur tend toutefois à s'estomper avec l'essor de procédés intermédiaires entres ces deux approches.

Débogage. 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.

Fuzzer. Outils qui servent à tester automatiquement des logiciels en leur présentant des données subtilement altérées à part de données correctes, afin d'essayer de produire dans ces logiciels des comportements incorrects. Ils permettent notamment de rechercher certains trous de sécurité. AFL (*American Fuzzy Lop*, du nom d'une race de lapins) est un exemple de *fuzzer*.

Langage machine. C'est le code numérique directement exécutable par le processeur de l'ordinateur. Celui-ci diffère d'une architecture de processeur à l'autre : par exemple, un Mac à processeur M1 ou M2, architecture ARM, ne peut pas exécuter directement des programmes écrits pour un Mac plus ancien, architecture Intel : il a besoin d'un système de traduction.

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.

Logique de Hoare. Un procédé utilisant des formules mathématiques comme annotations sur des programmes, ainsi que des règles pour passer d'une formule à l'autre, destiné à permettre de prouver que le programme vérifie une spécification mathématique. Elle doit son nom à C.A.R. Hoare (« Tony Hoare »).

Mémoire vive. Mémoire de travail d'un ordinateur, plus rapide que les stockages tels que les disques durs ou les mémoires flash.

P versus NP. Question liée à la complexité informatique, qui peut croître de façon polynomiale (P) avec la taille du problème ou de façon non polynomiale (NP) beaucoup plus rapide (souvent exponentielle).

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 celle de 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.

Alan Turing. Mathématicien britannique (1912-1954), auteur de travaux en théorie du calcul, en modélisation mathématique de la biologie, ou encore sur la question de l'intelligence artificielle. Il a, par ailleurs, pendant la Seconde guerre mondiale, conduit des travaux de première importance sur le décodage des transmissions chiffrées allemandes. En raison de son homosexualité, il a été condamné à subir une castration chimique, un « traitement » à base d'hormones femelles de synthèse. Sa mort, survenue deux ans après cette condamnation, est généralement considérée comme un suicide.

Figures

Figure 1. Explosion du vol inaugural de la fusée Ariane 5, en raison d'un bug informatique. © Agence spatiale européenne.

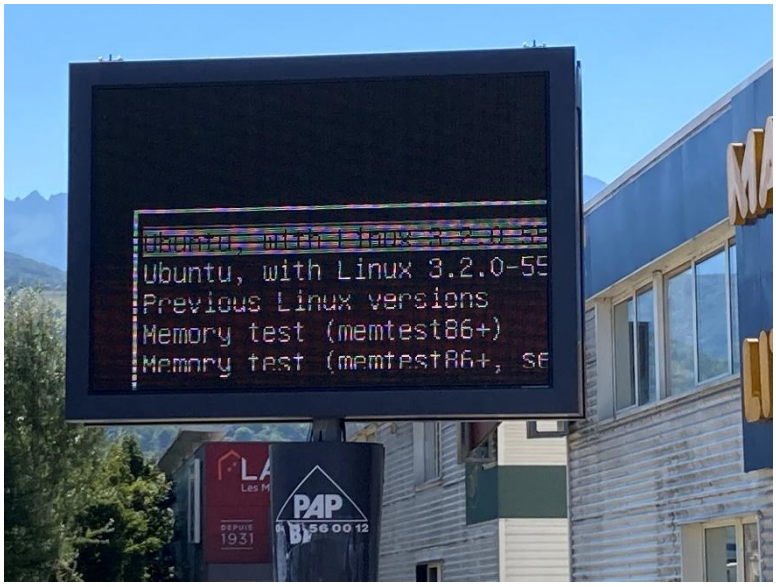

Figure 2. Des ordinateurs se cachent dans des appareils ménagers, des automobiles… ou des panneaux publicitaires. Ici, l'ordinateur de commande de l'affichage est en train de redémarrer. © David Monniaux.

```
int tampon[10];

void copier(int *t) {
  for(int i=0; i<=10; i++) tampon[i]=t[i];
}
```

Figure 3. Le programme suivant contient une erreur de programmation élémentaire, mais qui peut passer inaperçue pendant longtemps et créer par la suite des problèmes mystérieux.

Bibliographie

Cours de Xavier Leroy au Collège de France.

Alan Turing, *Checking a large routine*, 1949. (Premier exemple de preuve mathématique sur un programme informatique.)

Affiliation

David Monniaux. Directeur de recherche au CNRS, informatique, directeur du laboratoire Verimag à Grenoble (CNRS / Université Grenoble Alpes, Grenoble-INP). David.Monniaux@univ-grenoble-alpes.fr