Comment s'assurer qu'un programme informatique a été correctement écrit ? Lorsque la fiabilité est indispensable pour éviter des accidents, on fait appel à une approche située entre l'informatique et les mathématiques.

Vérifier des programmes en prouvant des théorèmes

Renaud Sirdey
est ingénieur de recherche chez Nortel.
renauds@nortel.com

Lors de son premier tir, la fusée Ariane 5 a explosé. Cause : erreur de programmation. Durant la première guerre du Golfe, un missile Patriot s'est écrasé sur un camp militaire américain. Cause : erreur de programmation. Le 15 janvier 1991, c'est le réseau téléphonique longue distance américain qui fut hors service. On pourrait ainsi multiplier les exemples. La cause est toujours la même…

Dans tous ces exemples, nous avons affaire à des systèmes dits à « logiciel prépondérant » : la mission confiée au logiciel est critique, et l'erreur de programmation a des conséquences dramatiques. Or, de telles erreurs sont inévitables. Alors, comment éviter l'inévitable ?

Comme l'a fait remarquer l'informaticien néerlandais Edsger Dijkstra dans les années 1970, ce n'est pas en testant un programme que l'on peut prouver qu'il ne contient pas d'erreur. En effet, exception faite des cas les plus simples, il n'est pas possible de vérifier le comportement d'un programme pour l'intégralité des valeurs pouvant être prises par ses paramètres : un programme qui prend comme paramètres deux entiers codés sur 32 bits* a à peu près autant d'entrées possibles qu'il s'est écoulé de microsecondes depuis le Big Bang, soit la bagatelle de quelques milliards de milliards ! Alors que faire ? Qu'est-ce, pour un programme, qu'être correct ? Et, surtout, comment établir qu'un programme est bien correct ?

Pré- et post-conditions

L'une des principales approches a été imaginée dans les années 1960 par le Britannique Tony Hoare (en médaillon), l'Américain Robert Floyd et le Danois Peter Naur. Leurs travaux ont été depuis prolongés et unifiés, entre autres, par le Français Jean-Raymond Abrial. L'idée est qu'un programme n'est correct que par rapport à une expression de ce qu'il doit faire, c'est-à-dire par rapport à une *spécification.* Prenons l'exemple d'une procédure supposée calculer la racine carrée d'un nombre positif ou nul : on lui donne un nombre $x \geq 0$ et elle fournit un nombre y tel que $x = y \times y$. Tout est dit sur le quoi, rien sur le comment !

TECHNIQUE Un exemple de preuve de correction

ON SE PROPOSE D'ÉTABLIR la correction du programme élémentaire qui échange le contenu des deux variables x et y :
$P : x = a$ et $y = b$;
$S : t := x ; x := y ; y := t ;$
$Q : x = b$ et $y = a$.

LA PRÉCONDITION P exprime que x contient initialement la valeur a et que y contient initialement la valeur b ; la postcondition Q indique qu'après l'exécution du programme S, on souhaite que x contienne la valeur b et que y contienne la valeur a.

ON COMMENCE PAR CALCULER $[t := x ; x := y ; y := t](x = b$ et $y = a)$ à l'aide des axiomes d'affectation et de composition :
$[t := x ; x := y ; y := t](x = b$ et $y = a) \equiv [t := x]$
$[x := y][y := t](x = b$ et $y = a)$
or,
$[y := t](x = b$ et $y = a) \equiv (x = b$ et $t = a)$;
$[x := y](x = b$ et $t = a) \equiv (y = b$ et $t = a)$;
$[t := x](y = b$ et $t = a) \equiv (y = b$ et $x = a)$.

ON DOIT DONC PROUVER LE THÉORÈME $P \Rightarrow [S]Q$ sachant que, dans ce cas très simple, on a les relations d'équivalence : $P \equiv (x = a$ et $y = b) \equiv (y = b$ et $x = a) \equiv [S]Q$ selon notre calcul.
Le programme est donc correct.

MÊME LES MEILLEURS PROGRAMMEURS (comme ceux réunis ici par Google en 2006 pour une compétition d'informaticiens) peuvent faire des erreurs. Dès lors, la vérification de la justesse des programmes est confiée à l'ordinateur pour la majeure partie des preuves requises. © RAMIN TALAIE/CORBIS

Pour spécifier un programme, autrement dit pour formellement exprimer ce qu'il doit faire, Tony Hoare a introduit les notions de « précondition » et de « postcondition », qui correspondent à des conditions qui doivent être vérifiées respectivement par les variables d'entrées et de sortie du programme (après son exécution). Un programme est correct, *primo* s'il finit par s'arrêter, et *secundo* si, dès lors que la précondition ($x \geq 0$ pour l'exemple du calcul de la racine carrée) est vérifiée au début de son exécution, la postcondition ($x = y \times y$) est vérifiée à la fin. On définit le programme comme un triplet composé d'une précondition P, d'un ensemble d'instructions S et d'une postcondition Q.

Des axiomes intuitifs

Pour montrer qu'un programme est correct, on s'intéresse à la précondition la plus « faible », notée $[S]Q$: c'est la condition la moins restrictive garantissant la satisfaction de Q après l'exécution de S. Si l'on a la relation $P \Longrightarrow [S]Q$, autrement dit si P est au moins aussi restrictive que $[S]Q$ – par exemple la condition $x \geq 1$ est plus restrictive que $x \geq 0$ car $[1,+\infty[$ est un sous-ensemble de $[0,+\infty[$ – alors le programme est correct. De plus, $[S]Q$ se déduit « mécaniquement » – un ordinateur peut le faire – de S et de Q à l'aide d'une série d'axiomes que l'on appelle aujourd'hui « axiomatique de Hoare ».

Prenons deux exemples : l'axiome d'affectation et l'axiome de composition. On définit x comme l'ensemble des variables d'un programme et e un ensemble de valeurs que peuvent prendre ces variables. L'axiome d'affectation stipule que pour que $Q(x)$ soit vrai après l'exécution du programme composé de l'unique instruction « x prend la valeur e » écrite « $x := e$ », alors il faut que $Q(e)$ soit vrai avant l'exécution du programme. Cela se note $[x := e]Q(x) = Q(e)$. Cet axiome est intuitif. Par exemple, si $Q(x)$ est $x \geq 0$, pour que $Q(x)$ soit vrai après l'exécution de « $x := 3$ » alors il faut que $Q(3)$ (soit $3 \geq 0$) soit vrai avant.

L'axiome de composition, quant à lui, dit que pour que Q soit vrai après l'exécution du programme composé de l'instruction S suivie de l'instruction T, alors il faut que la postcondition obtenue après exécution de S corresponde à la précondition la plus faible selon laquelle T établit Q. Cela se note $[S ; T]Q = [S][T]Q$. Là encore cet axiome est très intuitif : par exemple pour que $Q(y)$ soit vrai après l'exécution du programme « $x := 3 ; y := x$ », il faut que $Q(x)$ soit vrai avant l'exécution de « $y := x$ », c'est-à-dire après l'exécution de « $x := 3$ », et donc que $Q(3)$ soit vrai avant l'exécution de cette dernière instruction (lire l'encadré « Un exemple de preuve de correction »).

Il existe d'autres axiomes, plus complexes, pour traiter les cas des constructions « si E alors S sinon T » et « tant que E faire S ». Cette dernière construction est toutefois plus délicate à traiter en raison de l'indécidabilité du problème de l'arrêt*.

Créativité du programmeur

Qu'avons-nous gagné, puisque, pour construire des programmes corrects, il faut maintenant construire des preuves correctes ? Il existe en réalité une différence fondamentale entre programmes et preuves : si les programmes ne peuvent pas être vérifiés mécaniquement – par l'ordinateur –, les preuves, elles, peuvent l'être. Il suffit en effet de s'assurer que chaque étape du raisonnement consiste en l'application d'une règle de déduction valide. La seule condition est qu'elles soient exprimées de manière suffisamment précise, c'est-à-dire dans un langage formel compréhensible par la machine. Nous avons donc gagné que le couple programme/preuve apporté par l'homme peut être vérifié automatiquement par l'ordinateur.

Le principal frein à une application à grande échelle de l'approche imaginée par Hoare est sa lourdeur : pour établir la correction d'un programme complexe, il convient généralement de prouver une multitude de « petits » théorèmes (les « $P \Longrightarrow S[Q]$? » pour chacune des procédures du programme). Le problème réside donc plus dans le nombre élevé de preuves que dans la difficulté de ces dernières.

De nos jours, cette difficulté est contournée par l'utilisation de « prouveurs » de théorèmes interactifs. L'ordinateur, à l'aide d'heuristiques, décharge le programmeur de la majeure partie des preuves requises, mais fait appel à sa créativité lorsqu'il se retrouve dans l'incapacité d'établir un théorème. Le programmeur, devenu mathématicien, complète alors la preuve ou conclut que le théorème est faux, ce dernier cas étant symptomatique d'une erreur de programmation ou, plus exactement, d'une incohérence entre le programme et sa spécification. ■■

* Un **bit** est l'unité d'information en informatique, huit bits formant un octet. Il prend la valeur 0 ou 1.

* Le **problème de l'arrêt** consiste à dire si un programme qui effectue un calcul sur un nombre s'« arrêtera » ou se perdra indéfiniment en calculs.

POUR EN SAVOIR PLUS

Jean-François Monin, *Introduction aux méthodes formelles*, Hermès, 2000.