Interprétation quantique du logiciel

François Pessaux

Institute of Useless and Dummy Theories
42 rue Dev Null, 07700 Bidon, France
firstname_lastname@yahoo.fr

1 Introduction

Depuis des décennies, les évolutions en informatique se catégorisent en principalement en deux parties: d'une part les travaux réalisés en vue d'obtenir des fonctionnalités de la part d'un système programmé, d'autre part ceux réalisés en vue d'éliminer les erreurs de programmation introduites dans ces systèmes.

Dans cet article, nous tentons de démontrer que le second point, sur lequel nombre de méthodes se heurtent à des problèmes de complexité et d'indécidabilité, n'est en fait dû qu'à une mauvaise interprétation, une mauvaise appréhension de la nature d'un programme informatique. Les contributions majeures de cet article sont la présentation de cette nouvelle vue de l'informatique, ainsi que la proposition une solution efficace pour se débarrasser des erreurs dans les programmes.

2 Du calcul au quantique

Depuis les débuts des travaux sur la calculabilité puis sur son application à la réalisation de systèmes de calcul automatique, le logiciel a toujours été considéré comme le déroulement systématique d'une séquence d'instructions par un organe de calcul: le microprocesseur. Le programme était donc considéré comme une forêt de chemins d'exécution parcourue par le microprocesseur, à chaque instant ce dernier exécutant une instruction faisant partie d'un de ces chemins. Une erreur dans un programme se caractérise par le parcours d'un chemin incorrect, soit au niveau de la sémantique que l'on espère du programme (on obtient alors un résultat erroné), soit au niveau "instruction machine" (le microprocesseur rencontrant une instruction mal formée, ne peut pas l'exécuter). Dans les deux cas, l'exécution "plante": le résultat n'est pas celui escompté.

Pratiquement, les erreurs revêtent de nombreuses formes aboutissant à des défaillances reproductibles, aléatoires, intermittents ou non-reproductibles. Ceci les rend difficiles à analyser.

En 1915, Albert Einstein présenta la théorie de la relativité générale [1]. L'analyse fine de cette théorie nous donne déjà une piste de réflexion: le "bon" fonctionnement d'un programme est, comme tout, relatif. Cela dit, les effets relativistes ne se manifestant qu'à des vitesses proches de celles de la lumière, il est peu probable qu'ils n'expliquent la présence d'erreurs dans les programmes.

L'autre grande révolution de la physique moderne, la mécanique quantique apparue autour de 1930 grâce aux travaux de Dirac, Heisenberg[3], Schrödinger[5] pour ne citer qu'eux, nous apporte une vue plus claire de l'informatique.

Dans notre approche, contrairement à l'informatique classique où un programme "suit" un chemin d'exécution, nous postulons que ce programme, lors d'une exécution, passe par tous les chemins possibles à la fois. Nous transposons ainsi ici l'expérience des fentes de Young[7], aux programmes. Ainsi, comme l'équation de Schrödinger nous le décrit, nous n'avons plus *un* seul résultat possible pour une exécution d'un programme (**même déterministe**), mais une *probabilité* d'obtenir tel ou tel résultat. C'est la *fonction d'onde* du programme. Ainsi, l'état $|\phi\rangle$ d'un programme est:

$$\int \phi(x,t)|c\rangle dx$$

où ϕ est la fonction d'onde du programme. Ainsi, le programme:

```
int f () { return (42) ; }
```

ne retournera pas forcément la valeur 42, mais aura *un certain nombre de chances* de se comporter comme tel, en fonction de sa fonction d'onde. Il est ainsi clair, que ceci explique simplement les bugs non-reproductibles et aléatoires, la fonction d'onde devant ne pas être du côté de l'utilisateur à ces moments-là.

Ainsi, tout comme dans le paradoxe du chat de Schrödinger[5], le programme est dans une superposition d'états: tant qu'aucune observation est réalisée, il rend tous les résultats possibles. Ainsi, tant qu'aucune observation n'est effectuée, un programme donne toujours le résultat correct, le résultat attendu.

Tout comme l'observation d'une particule nécessite son "éclairage" par une onde, l'observation d'un programme nécessite que l'utilisateur "l'éclaire". Le principe d'incertitude vient ici jouer son rôle: plus on cherche précisément la position d'une particule, plus il faut "l'éclairer" avec un faisceau de petite longueur d'onde, donc énergétique. Il en résulte une perturbation de la position de la-dite particule. On ne peut donc connaître précisément à la fois la vitesse et la position de la particule. Il en est de même pour un programme: il on ne peut connaître précisément ces 2 caractéristiques. Si l'on augmente notre connaissance de la vitesse à laquelle il tourne, alors on perd en connaissance de sa position, donc des valeurs qu'il a calculé à cet instant. Inversement, si l'on s'intéresse uniquement à ses résultats, alors il faut perdre de la précision quant à la vitesse à laquelle il tourne. On pourrait se dire que le plus important est justement sa position, i.e. son (ses) résultat(s). Mais, si l'on ne sait pas précisément à quelle vitesse il tourne, on ne peut être sûr que sa vitesse n'ait jamais été toujours nulle, donc que le programme ait tournée un jour ! Comment faire confiance à un résultat s'il n'a jamais été calculé ?

Lorsque l'on s'intéresse au programme au niveau microscopique, les manifestations quantiques ne peuvent plus être ignorées. Effectivement, un programme ne peut changer d'état que par intervales discrets: les *quantas*. Ainsi, si le programmeur n'a pas pris soin de calibrer ses instructions comme de la taille d'un quantum, le programme sautera au milieu d'une instruction et donc

se plantera. Pour autant, tout comme la mécanique quantique autorise à "emprunter de l'énergie" tant qu'elle est "remboursée" suffisamment vite (d'autant plus d'énergie qu'elle est retournée sur un laps de temps plus petit), un programme peut en cours d'exécution, prélever une somme d'argent sur un compte temporairement tant qu'il le rembourse. Si par malheur, un bug se produit entretemps (par exemple, un utilisateur perturbe la superposition quantique de tous les états du programme – dont celui représentant le résultat correct– en regardant son écran-, le programme s'arrête, n'a pas eu le temps de rembourser, et ainsi se déclenche une crise boursière, ou un prétendu détournement de fonds.

Il découle de cette analyse un intérêt majeur en terme d'enseignement dans les facultés et grandes écoles. Puisque le résultat de l'exécution d'un programme est totalement aléatoire, il n'est plus nécessaire de faire perdurer l'enseignement de certaines matières telles que la compilation et la sémantique. Il est à noter que, malgré le fait que nous soyons les premiers à proposer une interprétation quantique du logiciel, et donc de mettre en avant la raison motivant la suppression de ces matières, force est de constater que certaines entités d'enseignement spécialisées en informatique ont d'ores et déjà commencé à mettre en pratique cette modification des enseignements.

3 Une solution élégante

À la lueur de la précédente description, la solution est immédiate (une version étendue de ce papier fournira en annexe la preuve formelle en Coq[6] de la validité du raisonnement). Vu que c'est l'observation qui introduit la rupture de superposition, vu que ce sont les utilisateurs qui provoquent majoritairement ces observations, l'évitement d'utilisateurs permet *de facto* la suppression des comportements indésirables, donc des erreurs dans les programmes. Ainsi, nous préconisons de supprimer les utilisateurs de logiciels.

Une question épineuse reste: pourquoi ne pas non plus supprimer les développeurs puisque lorsqu'ils testent (vite et mal) leur programme, il l'observent ? La réponse est qu'il n'est pas nécessaire de supprimer les développeurs, supprimer les tests qu'ils faisaient suffira.

Reste alors que l'on pourrait se demander à quoi servent les développeurs et donc pourquoi derechef ne pas les supprimer. La réponse à cette question fondamentale est que, en temps que chercheur en informatique, vous lecteur, moi, nous sommes amenés parfois à développer des morceaux de logiciel... et vous, nous, ne voudrions quand même pas nous retrouver au chômage.

D'éminents collègues se sont posé des questions fondamentales à propos de nos conclusions.

Citons FRS[2]: "*La question fondamentale est: si les tests aussi sont automatisés, et lancés automatiquement sans intervention humaine, alors l'incertitude quantique des tests compense-t-elle celle du code, ou bien s'y ajoute-t-elle ?*". La réponse la plus probable est que pour ce qui est des tests, ils sont soumis à la même incertitude quantique. Donc il est possible de les lancer, mais il ne faut pas les observer sous peine de faire s'effondrer la superposition quantique et

donc de modifier l'expérience et donc, possiblement, provoquer une orientation vers un état où le test est KO alors que sans l'observer il aurait été OK. Donc non, il n'y a pas "compensation" de l'incertitude. Par contre, y a-t-il addition ? Ajouter de l'incertitude à de l'incertitude... cette question reste ouverte, rien n'est certain.

De même, citons PHA[4] qui proposait: "*Au lieu de supprimer les tests (ce qui serait mal venu dans cette période de recherche du plein emploi), il serait judicieux de n'écrire et de n'exécuter que les tests qui permettent d'identifier un comportement non désiré (les autres tests ne permettant pas de découvrir quelque chose). Cette optimisation permet d'avoir une efficacité maximale des tests à moindre coût.*". Malheureusement, rien que le fait d'observer le résultat de ces tests perturberait leurs résultats. D'où, comment avoir la certitude que cette observation ne les a pas fait acquérir un état stable OK alors qu'ils auraient dû être KO simplement parce que le testeur aurait fait une erreur, non pas dans le logiciel, mais dans sa sélection des tests (éventuellement en utilisant un logiciel de tests, lui-même soumis à une modélisation quantique et réciproquement dans son entière territorialité). Inversement, il serait possible que les tests en se montrent KO alors qu'ils auraient dû être OK simplement du fait de la perturbation observationnelle, ce qui masquerait alors le fait que le test n'aurait pas dû être passé puisque normalement OK (par hypothèse de la proposition, on ne devrait passer que les tests menant à KO).

4 Conclusion et travaux futurs

La solution raisonnable exposée ici consiste donc à laisser l'informatique aux informaticiens, chose déjà largement en vogue puisque à chaque problème sous son Wi..., pardon, OS favori, M^e et M^r finissent toujours par faire appel au dépanneur "professionnel" ou au copain du fiston qui est "vachement en informatique". De plus, le domaine ne devenant ainsi plus que confiné aux sachants du domaine, qui tous tiennent à leur travail, ils ne manqueront pas de se serrer les coudes lors de la rédaction des rapports pour les autorités qui les payent, afin de justifier de l'intérêt de leur travaux et pouvoir continuer en paix à faire tout et n'importe quoi.

Les présents travaux nous suggèrent de nouvelles pistes d'exploration, comme la cartographie systématique du génome informatique, qui comme tout le monde le sait comporte principalement 2 acides aminés: la 0-ine et la 1-ine. Ces travaux pourraient à terme permettre la création de cellules souches informatique en vue de réparer, par culture *in vitro*, des morceaux erronés de programmes dans le cas où la recommandation proposée dans le présent papier ne serait pas appliquée.

References

1. Albert Einstein. La théorie de la relativité restreinte et générale. *Édition française Gauthier-Villars 1956*, 1916.

2. S FR. De quantum principia ex nihilo aleacta jacta est non laudat. *Brain Excercices and Computation*, 2011.
3. Werner Heisenberg. Les principes physiques de la théorie des quanta. *Gauthier-Villars. Réédition par Jacques Gabay (1989) ISBN 2-87647-080-2*, 1932.
4. A PH. Epistémiologie de la géobiologie quantique abstraite. *Journal of applied fuzzy paradigms*, 2011.
5. Erwin Schrödinger. La situation actuelle en mécanique quantique, article dans lequel apparait le "chat de schrödinger" pour la première fois. 1935.
6. The Coq Development Team. The Coq Proof Assistant Version 8.0, July 2004. http://coq.inria.fr/.
7. Thomas Young. Course of lectures on natural philosophy and the mechanical arts. *London: Taylor and Walton*, 1845.