Les mathématiques à la conquête d'un logiciel fiable

Anne-Marie Roy-Boulard

Le développement incessant de la technologie informatique nous permet d'en faire une utilisation de plus en plus active. Effectivement, nous sommes pratiquement devenus dépendants du moteur central des systèmes informatisés, les logiciels ! Si leur utilisation nous est particulièrement bénéfique, leur conception ne suit généralement pas de logique formelle et leur fiabilité est douteuse. Vous pensez actuellement à votre logiciel de traitement de texte ? Rien n'est bien alarmant. Imaginez plutôt les conséquences d'une erreur non détectée dans le logiciel qui contrôle la circulation aérienne ou encore celui qui contrôle un appareil médical. Heureusement pour nous, les concepteurs de logiciels peuvent maintenant utiliser des concepts mathématiques pour améliorer leur travail.

Afin de réaliser les tâches pour lesquelles il a été conçu, un système informatique reçoit des instructions en provenance de son logiciel de contrôle. Celui-ci est comparable à un plan d'assemblage où toutes les instructions sont reliées entre elles pour donner le produit désiré. Par exemple, lorsqu'une distributrice vous sert un café, c'est qu'elle en a reçu la commande dans son langage de machine. Toutefois, des erreurs peuvent s'introduire lors de la conception générale du logiciel ou dans l'écriture du code. Voilà que se pointent les problèmes imprévus ! Vous pourriez donc avoir droit à un chocolat chaud plutôt qu'au café que vous aviez commandé. Et si les risques d'intrusion d'erreurs sont faibles pour les logiciels peu complexes, comme dans une distributrice à café, nous devons toutefois porter une attention particulière sur certains logiciels tombant dans la catégorie dite « à lourdes conséquences ». Ces derniers comportent en effet des milliers de lignes d'instructions. Aussi, les mathématiques discrètes se révèlent être fort utiles pour contrer les risques d'erreurs plus élevés et pour vérifier l'exactitude de leur conception [1].

Nombre de lignes d'instructions dans certains logiciels

Rasoir :	25
Moteur d'une automobile :	30 000
Système téléphonique :	15 000 000
Navette spatiale :	20 000 000

Bien loin de la pensée mathématique

Pour produire un logiciel, le programmeur s'informe des exigences de son client et se met directement à l'ouvrage. Assis devant son ordinateur, il écrit en langage informatique les instructions nécessaires pour obtenir le fonctionnement désiré. Le programmeur n'a pas à se soucier de normes de constructions et la transformation d'une tâche spécifique en instructions compréhensibles par l'ordinateur n'est soumise à aucune rigueur scientifique. Elle n'est limitée que par l'imagination et l'expérience du concepteur.

Contrairement à un objet physique, il est très difficile de visualiser concrètement un logiciel de sorte qu'il est impossible de certifier que le code écrit ne produira pas d'erreurs. Cette approche sans exigence méthodique ne permet la vérification du logiciel qu'une fois sa conception entièrement terminée. Le programme est alors testé à maintes reprises afin d'observer ses comportements et de voir s'il fonctionne adéquatement [4]. Remarquez que cette façon de faire ne peut pas détecter toutes les erreurs ! Il nous faudrait essayer toutes les combinaisons possibles d'événements ainsi qu'imaginer toutes les surprises de la vie, ce qui est pratiquement infaisable.

Exemples d'événements causés par des erreurs non détectées dans des logiciels

- 1981 : dans un hôpital du Texas, un logiciel de contrôle des rayons X tue deux personnes en leur donnant une décharge de radiations.
- 1990 : 50 millions d'abonnés d'AT&T voient leurs appels interurbains se perdre dans le néant pour des raisons mystérieuses.
- 1992 : aux États-Unis 165 millions $ sont perdus lors de l'annulation d'un projet qui devait associer les réservations aériennes, les réservations d'automibiles Budget et les réservations des hôtels Marriot et Hilton [2].
- 1994 : le satellite *Clémentine* connaît 11 minutes de délinquance, ce qui l'empêche de remplir une de ses missions.
- 1994 : 44.3 millions $ sont gaspillés en Californie dans un logiciel qui devait relier les systèmes d'enregistrements automobiles et des permis de conduire. Le projet est abandonné après sept ans de travail [2].

Penser, dessiner et prouver

Entre 70% et 80% des logiciels construits échouent parce qu'ils contiennent des erreurs incorrigibles et même parfois inexplicables [2]. Il est difficile d'accepter que de telles situations surviennent et que nous ne puissions les comprendre. Pour éviter que cela arrive, nous avons besoin de lignes de conduite et d'outils permettant de déceler les erreurs le plus tôt possible durant le développement. Considérant un logiciel comme étant une immense formule, l'idée d'employer la rigueur des outils mathématiques pour prouver formellement l'obtention de bons comportements s'est révélée fort utile. Ces outils mathématiques sont appelés les méthodes formelles.

Les méthodes formelles s'utilisent avant même de commencer à écrire le logiciel. Le concepteur du logiciel construit un plan de ce que sera le comportement de son système. Il utilise la logique mathématique ou un langage de spécifications à sémantiques formelles pour décrire ce comportement [4]. En examinant les propriétés du modèle ainsi conçu, les inconsistances logiques deviennent évidentes. Le prototype est aussi utilisé pour prouver que la conception du système satisfera les fonctionnalités désirées. Si des erreurs sont découvertes, les coûts de réparations sont moins considérables puisque les retouches sont faites seulement sur le croquis du logiciel. Cette étape de modélisation augmente la qualité du logiciel de façon significative [1].

Un automate est une structure mathématique utilisée en informatique. Il sert à modéliser des processus physiques qui changent d'état avec l'occurrence de certains événements.

Soit A un ensemble de symboles représentants ces événements. Un *automate* $\mathcal{A}$ est donné par un ensemble E d'*états* et une *fonction de transition* souvent partiellement définie

$$T : E \times A \to E,$$

qui, à chaque état et à chaque événement possible dans cet état, associe le nouvel état dans lequel se retrouvera l'automate. T(e, a) est souvent noté $e \cdot a$.

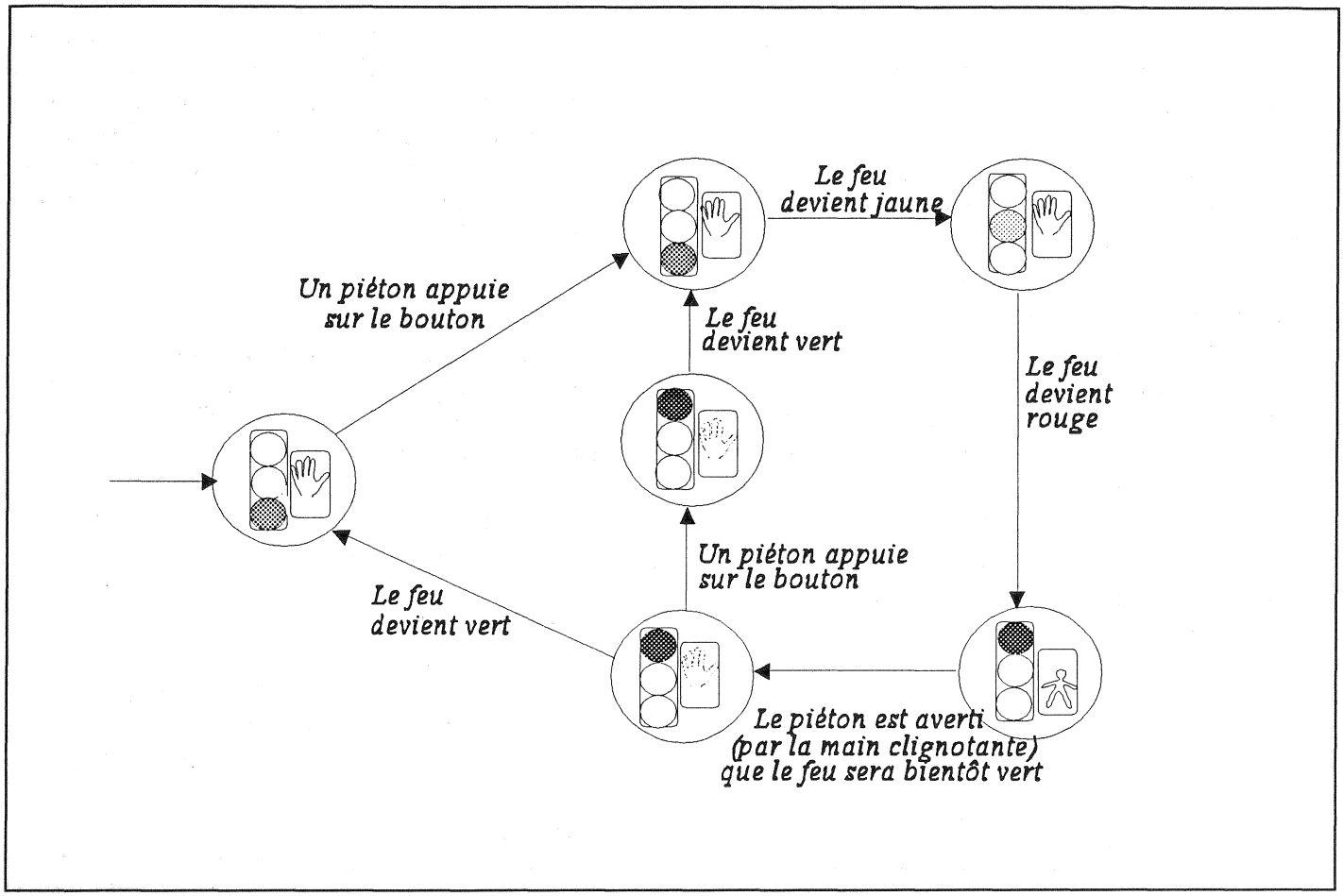
On désigne un état particulier e_0 de E comme étant l'état *initial* de l'automate et un ensemble de $F \subseteq E$ comme étant les états finaux. On dit qu'une suite d'événements

$$x_1 x_2 \ldots x_n$$

est acceptée par l'automate si

$$((e_0 \cdot x_1) \cdot x_2) \ldots \cdot x_n \in F$$

Ainsi, pour donner un exemple, modélisons un feu de signalisation à un passage pour piétons. Ce type de feu doit être vert en permanence sauf lorsqu'un piéton fait une requête pour traverser en appuyant sur le bouton à cette fin. La première étape consiste à définir ce que le système peut faire, c'est-à-dire les suites d'événements pouvant être exécutées par celui-ci. Partons du fait que le feu est initialement vert, le premier événement pouvant faire changer d'état ce système est l'arrivée d'un piéton appuyant sur le bouton. Éventuellement, les événements suivants seront *le feu devient jaune* puis *le feu devient rouge*. Au moment même où le feu devient rouge, le piéton reçoit le droit de traverser par le signal du « petit bonhomme qui marche ». Suite à cela, le feu doit demeurer rouge mais le piéton doit recevoir l'avertissement qu'il passera bientôt au vert.



Cet avertissement est donné par la « main clignotante ». Finalement le feu passera de nouveau au vert et la main redeviendra stable. Toutefois, lorsque la main clignote, un autre piéton peu appuyer sur le bouton de requête, le feu passera alors au vert tout en retenant qu'il devra éventuellement passer au jaune.

Cette étude des suites possibles d'événements peut être modélisée par un automate qui est un graphe dont les sommets sont des états du système et dont les flèches sont des événements. Ce graphe implique que notre système ne contient pas d'inconsistances logiques puisqu'on ne risque pas d'être emprisonné dans un quelconque état en ce sens qu'il est toujours possible de changer d'état.

Maintenant que les suites logiques d'événements possibles sont connues, on remarque qu'il serait important de considérer des délais de temps entre ces événements. Par exemple, l'événement *le feu devient rouge* ne doit pas arriver trop vite après l'événement *le feu devient jaune* puisque les automobilistes n'auraient pas suffisamment de temps pour immobiliser leur véhicule. Il importe donc de définir des contraintes temporelles sur les événements de façon à les empêcher ou les obliger d'arriver avant un certain délai. Les contraintes temporelles données par les ingénieurs civils sont les suivantes :

1) Le temps d'attente des automobilistes : le délai entre *le feu devient rouge* et *le feu devient vert* est $< D_0$.

2) Le temps d'attente des piétons : le délai entre *un piéton appuie sur le bouton* et *le feu devient rouge* est $< D_1$.

3) Le feu reste vert assez longtemps pour que les autos puissent passer : le délai entre *le feu devient vert* et *le feu devient jaune* est $> d_0$.

4) Les piétons ont assez de temps pour traverser : le délai entre *le piéton est averti que le feu sera bientôt vert* et *le feu devient vert* est $> d_1$.

5) Les automobilistes ont assez de temps pour s'arrêter : le délai entre *le feu devient jaune* et *le feu devient rouge* est $> d_2$.

Les ingénieurs auraient pu nous donner des valeurs réelles au lieu des paramètres (par exemple, le feu doit être vert pendant deux minutes). Par contre, il est beaucoup plus réaliste de considérer des paramètres généraux et ainsi trouver l'ensemble de toutes les valeurs possibles pour lequel le système est consistant. Dans un tel cas si un ingénieur revient sur sa décision en changeant la valeur d'un paramètre, on pourra immédiatement réajuster le système sans recommencer l'analyse au début. Finalement, une analyse par automates [5,6,7] du système décrit ci-dessus nous donne les valeurs possibles suivantes pour les paramètres :

$$D_0 > D_2 + d_1$$
$$D_1 > d_0 + d_1 + d_2$$

où D_2 est un nouveau délai, obtenu lors de l'analyse, entre les événements *le feu devient rouge* et *le piéton est averti que le feu sera bientôt vert.* Le système de signalisation routier peut donc être conçu physiquement et sera consistant si les concepteurs du logiciel respectent les conditions obtenues.

Au moment d'écrire les instructions du logiciel, le modèle nous est encore bénéfique. Ce dernier nous sert de marche à suivre. Après la rédaction du code, les méthodes formelles sont utilisées sous une autre forme. Il est effectivement possible de prouver que le programme est bien construit en faisant des preuves formelles de chacune des parties du logiciel. De telles preuves d'analyse sont produites surtout dans les cas où une très haute fiabilité est nécessaire. À ce stade, un logiciel a très peu de chance de contenir des erreurs. Le hic c'est qu'on est jamais sûr à 100% puisqu'une erreur peut avoir été faite en prouvant ou en décrivant le modèle de base. Finalement, des essais sur le produit sont exécutés. Après toutes ces précautions, il ne reste qu'à croiser les doigts... mais moins fort que si nous n'avions pas utilisé les méthodes formelles.

Entrer dans la cadence...

Une méthode formelle, appelée « B », a été utilisée en France pour améliorer le logiciel de contrôle de vitesse et de changement de voie qui guide 6 000 trains électriques [2]. Il en coûta 350 millions $ pour réduire la distance entre chacun des trains en augmentant leur vitesse. Un investissement considérable, mais grâce à celui-ci une économie d'un milliard $ a été réalisée en supprimant la construction d'une nouvelle ligne.

En voyant une telle réussite lors de l'utilisation d'un procédé, on pourrait croire que tous s'arrachent les méthodes formelles ... Eh non ! Avant de pouvoir vivre un tel engouement, les chercheurs en informatique théorique devront continuer à développer des outils pour appliquer ces techniques. Effectivement, les compagnies sont très réticentes devant l'utilisation des méthodes formelles. Ces méthodes étant d'une grande rigueur, elles demandent du temps et des gens hautement qualifiés pour les utiliser. Les concepteurs appliquant actuellement les méthodes formelles le font plutôt partiellement. Ils n'utilisent ces dernières que pour les parties jugées importantes.

Les chercheurs travaillent donc pour permettre une plus grande accessibilité à ces méthodes et espèrent qu'elles deviendront systématiquement utilisées pour le bien de tous. ■

Références bibliographiques

[1] A. Hall, (1990). *Seven Myths of Formal Methods*, IEEE Software.

[2] W. Gibbs, (1994). *Software's Chronic Crisis*, Scientific American.

[3] J. Bowen, M.G. Hinchey, (1995). *Ten Commandments of Formal Methods*, IEEE Computer.

[4] R.A. Kemmerer, (1990). *Integrating Formal Methods into the Development Process*, IEEE Software.

[5] A. Bergeron, (1995). *Symbolic Timing Devices*, Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, Lecture Notes in Computer Science No. 936, Springer Verlag, p. 504-517.

[6] A. Arnold, (1994). *Finite Transition Systems*, Prentice Hall.

[7] A. Bergeron, A.M. Roy-Boulard, (1997). *Contraintes temporelles paramétrisées et produits synchro-nisés,* Comptes rendus de la Conférence NOTERE'97, Pau, France (15 pages).

Anne-Marie Roy-Boulard
Laboratoire de Combinatoire
et d'Informatique Mathématique
Université du Québec à Montréal