Introduction aux méthodes formelles

Christian Rinderknecht

25 Octobre 2008

Introduction

- Les méthodes formelles sont des méthodes outillées pour le développement de logiciel qui font appel à des concepts d'informatique théorique.
- Les raisons de leur emploi sont en général : réduction des coûts de maintenance (en particulier corrective), robustesse (en délimitant explicitement le champ de validité du logiciel) et fiabilité.
- Leur domaine d'application est donc, en particulier, la sûreté de fonctionnement des systèmes critiques ou embarqués, car la correction et la continuité de service de ceux-ci peut mettre en jeu la vie d'êtres humains ou simplement d'importantes sommes d'argent.

La maîtrise de la complexité

- Il ne faut pas sous-estimer l'intérêt des méthodes formelles. Elles ont nécessité des dizaines d'années de recherche théorique ainsi que d'expérimentation et sont en pleine expansion.
- L'augmentation de la puissance de calcul des ordinateurs et la baisse de leur prix contribuent *de facto* à la pertinence de leur application et même parfois rendent leur application tout simplement possible dans certains domaines.
- Ce dont il est question ici, de façon indirecte, c'est la maîtrise de la taille et de la complexité logique des systèmes informatiques, qui est le thème central du génie logiciel.

Les limites de l'informatique

- Les inventeurs de mouvements perpétuels nous font sourire car leurs inventions contredisent nos connaissances fondamentales sur la physique : nous savons *a priori* qu'il y a une faille sans avoir besoin d'examiner le détail des mécanismes.
- En informatique il existe des réalisations qui doivent faire sourire. Mais lesquelles ? Devant le programme énorme qui fonctionne *presque*, que faire ?
  - Essayer de le compléter et de le corriger pour qu'il fonctionne correctement et complètement ?
  - Revoir la conception et recommencer avec une autre méthode de résolution du problème ?
  - Conclure avec amusement que le problème censé être résolu par le programme n'est en fait soluble par aucun programme, cette conclusion étant tirée sans examen détaillé du programme ?

Les limites et l'ingénieur

- Certaines limites à l'usage des méthodes formelles sont levées par des progrès techniques, mais certaines limites ne sont pas dues à l'électronique mais sont intrinsèques aux mathématiques, donc au calcul. Il est ainsi possible de démontrer que certains problèmes ne sont pas solubles et ne le seront jamais. La connaissance de ces limites extrinsèques et inhérentes fait donc partie des méthodes formelles.
- De nos jours, l'industrie du logiciel promeut fortement la « technologie à la carte » et la valse des mots-clés dans les *curriculum vitæ*. C'est pourquoi il est important pour l'ingénieur informaticien de comprendre les fondements de sa discipline, qui sont indépendants de toute technologie à la mode : il saura alors appréhender correctement les outils en fonction des concepts qu'ils impliquent.

Enjeux

- Les organismes publics ont pris conscience des progrès effectués. En Europe (et en France depuis 1995), les ITSEC (*Information Technology Security Evaluation Criteria*) exigent l'usage de méthodes formelles à partir d'un certain niveau de sécurité.
- En même temps qu'un élargissement des champs d'application (cartes à puces, systèmes d'information hautement sécurisés, robotique, commerce électronique, contrôle aérien etc.), on observe une montée des exigences.
- Dans certains cas (transports, pilotage de centrales nucléaires, informatique médicale) des vies humaines sont en jeu.

Enjeux (suite et fin)

- Dans les télécommunications, la panne d'un réseau d'ATT en janvier 1990 a causé des pertes économiques considérables.
- L'explosion du vol 501 de la fusée européenne Ariane en 1995 et l'échec d'un missile sol-air Patriot (provoquant de nombreux morts dans un camp militaire états-unien) sont tous deux dus à des problèmes logiciels et ont montré l'insuffisance des méthodes en vigueur face aux enjeux d'aujourd'hui.
- Actuellement, les méthodes formelles s'appliquent à des systèmes situés entre l'algorithme sophistiqué de quelques pages et le logiciel de plusieurs dizaines de milliers de lignes, *ce qui est déjà fort utile*.

Spécification pour la réingénierie

- Une expérience a été menée en 1990 entre IBM et l'université d'Oxford : une restructuration majeure dans un gros logiciel de gestion de transactions (800 000 lignes d'assembleur et d'un langage propriétaire de haut niveau).
- 268 000 lignes ont été modifiées ou réécrites, dont 37 000 ont été rédigées avec la méthode formelle Z. Des moyens de mesure ont été mis en place.
- Les coûts de développement ont diminué de 9%.
- Au cours des huit premiers mois suivant la remise en service, les clients ont recensé 2,5 fois moins d'erreurs sur les parties développées en Z par rapport aux autres et ces erreurs étaient perçues comme moins graves.

Preuve de logiciel ferroviaire critique

- Pour apporter des garanties forte de correction du logiciel il faut d'abord bien spécifier le comportement attendu et délimiter le champ d'application des preuves.
- L'utilisation de la méthode B par GEC-Alsthom et récemment par Matra Transport International dans le métro de Calcutta (1992) et la ligne 14 (1999) du métro parisien illustre cette approche.
- Cette méthode permet de raffiner pas à pas la spécification initiale et offre un outil (reposant sur une théorie des ensembles) pour prouver à chaque étape la conformité (correction) du raffinement par rapport à l'étape précédente. La dernière phase consiste à produire quelques dizaines de milliers de lignes de code C dont on possède alors par transitivité une preuve de correction par rapport à la spécification initiale.

Communication et rigueur

- Un intérêt souvent méconnu de la sémantique rigoureuse ou formelle est de faciliter la *communication* en constituant un cadre non ambigu au discours et un arbitre impartial. C'est aussi un excellent guide pour construire des outils de support (cf. les techniques de compilation). Il est aussi parfois possible d'énoncer alors les propriétés attendues d'un système et même d'en prouver certaines.
- N'oublions pas que la maintenance logicielle engloutit en moyenne les deux tiers du coût total d'un projet et que la réparation d'une erreur de spécification demande environ vingt fois plus d'efforts si elle n'est détectée que lors de l'exploitation.
- Les méthodes formelles introduisent une *rigueur* supplémentaire dans le génie logiciel pour réduire ces coûts.

Dans « méthode » formelle il y a « méthode »

- Le génie logiciel montre qu'il est important de consacrer un gros effort dans les premières étapes du cycle de vie et, particulièrement, d'établir des spécifications fiables, c'est-à-dire
  - correspondant bien à ce que l'on en attend intuitivement
  - et cohérentes.
- Le premier point repose sur une bonne connaissance des besoins du client, connaissance que ce dernier ne peut pas toujours exposer d'emblée. Une *maïeutique* doit alors s'établir entre lui et le concepteur pour dégager les propriétés du résultat attendu. Le concepteur construit pas à pas un modèle formel qu'il ne partage pas forcément avec le client mais dont il confronte avec lui les implications logiques dans le but d'affiner les propriétés.

Dans « méthode formelle » il y a « méthode » (suite et fin)

- Le concepteur peut aussi recourir au *prototypage rapide*, qui permet d'élaborer en peu de temps une version facilement modifiable du système qui semble souhaité. Les techniques employées doivent alors avant tout favoriser la vitesse de réaction du processus de développement.
- Les considérations relatives à la propreté ou à l'efficacité du logiciel peuvent s'avérer pénalisantes *à ce niveau*.
- Lorsque l'étape du prototypage est franchie, les priorités changent et les objectifs de qualité et de rigueur reviennent au premier plan.
- L'emploi de langages fonctionnels, comme ML, simplifie le passage du prototype au produit.

Du langage formel comme fil conducteur

- Formuler le problème à résoudre dans un langage formel ou semi-formel est la première étape de la solution.
- En effet, un langage formel peut être muni d'une sémantique sûre, surtout s'il est fondé sur des théories mathématiques éprouvées.
- Il devient possible de prouver des propriétés du système, d'une part, au niveau de la spécification, d'autre part, au niveau du code qui peut donc être garanti conforme à la spécification.
- La *correction* dit que toutes les valeurs retournées par le programme sont bien prévues par la spécification.
- La *complétude* dit que le programme n'en oublie aucune.

Du langage formel comme fil conducteur (suite et fin)

- Pour des raisons qui touchent aux fondements des mathématiques, ces propriétés, comme d'autres, peuvent ne pas être démontrables en général (on parle d'*indécidabilité*). Cela nous ramène donc aux limites de la calculabilité.
- Il y a plusieurs façon d'aborder la conformité (correction) : logique de Hoare, énumération des états atteignables, raffinage des spécifications, réécriture, calcul ou extraction de programmes.
- Un langage formel se prête bien à l'élaboration d'outils permettant d'assister automatiquement les différentes tâches.
- Les efforts de test, de maintenance et parfois de codage sont diminués et la maîtrise du cycle augmente, la documentation étant devenue bien plus fiable.

Les méthodes formelles pour le donneur d'ordre

- Les méthodes formelles concernent également les organismes qui font réaliser tout ou partie de leurs développements logiciels par des tiers. En effet, le donneur d'ordre doit s'assurer
  - que ses spécifications sont correctes,
  - que le produit livré correspond aux spécifications.
- Pour le second point, le donneur d'ordre doit au minimum valider le produit et, pour cela, il le soumet à une batterie de tests.
- La complexité des tests croît grandement avec celle du produit et de ses nouvelles versions.
- La production automatique de jeux de test à partir de spécifications formelles enregistre de plus en plus de succès.

Les limites du test pour le donneur d'ordre

- Le test, s'il est toujours nécessaire, peut se révéler insuffisant, selon le degré d'exigence, car il ne couvre qu'un certain nombre de comportements du système. La couverture structurelle de toutes les branches du code ne tient pas compte de toutes les combinaisons possibles de valeurs (qui peuvent être infinies en théorie).
- Parfois le donneur d'ordre fournit au réalisateur le jeu de test que doit réussir le produit. Cela peut se révéler problématique si le réalisateur fournit volontairement un produit vérifiant les tests (donc un nombre fini de cas) mais erroné par ailleurs.
- Si le produit est construit au moyen de méthodes formelles, il peut être livré avec une preuve de sa conformité, que le donneur d'ordre peut alors vérifier. Il est bien plus facile de vérifier une démonstration que de la découvrir.

Quelques bémols...

- Les méthodes formelles ne sont pas une panacée : face à des problèmes complexes il n'y a pas de solution de facilité mais des compromis raisonnés.
- Il faut garder à l'esprit la distance qui sépare la spécification formelle et la réalité qu'elle modélise. Elle ne peut être réduite que par une démarche de relectures, de reformulations, de confrontation à des conjectures, de maïeutique.
- Un obstacle à l'emploi des méthodes formelles est le manque de culture mathématique et de maîtrise des notations. En génie civil ou en électronique les mathématiques font partie du paysage.

... et bécarres

- Les méthodes formelles exploitent beaucoup les phases initiales du développement (spécification, conception) au risque de produire une impression négative de ralentissement.
- Mais les phases ultérieures (tests, intégration) sont raccourcies et mieux maîtrisées.
- La formalisation (c.-à-d. l'emploi d'une logique pour modéliser) révèle plus tôt nombre de points délicats.
- Ces points délicats sont souvent inhérents au problème à résoudre, qui peut donc être bien plus complexe qu'il n'y paraît au premier abord.
- L'usage de notations abstraites est alors en fait une simple nécessité, pas une complication adventice.

Tour d'horizon des méthodes formelles

Les approches formelles présentent des aspects très variés. Il en existe actuellement plusieurs familles, la plupart comportant

- une théorie sous-jacente particulièrement mise en avant (par exemple les systèmes de transitions, les théories des ensembles, les algèbres universelles, le λ-calcul),
- un domaine de prédilection (par exemple les traitements de données, le temps-réel, les protocoles)
- et une communauté de chercheurs et d'utilisateurs, parfois répartis en plusieurs variantes ou écoles.

Approches spécialisées ou générales

- La spécification d'un système recouvre plusieurs aspects dont l'architecture, les comportements observables, les bases de données, les algorithmes etc.
- Certaines méthodes modélisent les systèmes sous l'angle de la transformation de données, ou de flux, ou de machines à états.
- Les échanges d'information peuvent être modélisés par partage de données (mémoire centralisée), par communication synchrone (suppose une horloge unique sans délai de propagation) ou asynchrone de messages, par invocation de fonctions (type *Remote Procedure Call* en C ou *Remote Method Invocation* en Java).
- D'autres méthodes sont moins contraignantes et fournissent un excellent cadre interprétatif général.

Approches spécialisées ou générales (suite)

- Les approches spécialisées, en privilégiant souvent l'aspect opérationnel (outillage), peuvent obscurcir la compréhension d'ensemble et suivre difficilement les évolutions de l'objet étudié.
- Les approches générales, plus proches de la logique mathématique, offrent une grande liberté d'expression mais pêchent sur le plan méthodologique.

Exemple : le rôle des états

- Un exemple typique de différence d'approche est le rôle accordé aux états d'un système dans sa modélisation.
- Cette notion semble fondamentale si l'on considère le support d'exécution des logiciels (ordinateurs ou machines virtuelles) qui suppose une mémoire dont le contenu peut changer et, éventuellement, une horloge.
- D'un autre côté, cette notion n'est pas fondamentale en mathématiques, bien que les mathématiques peuvent parler d'états. D'un point de vue théorique, il est plus délicat de décrire la composition d'intructions avec des effets sur la mémoire que de fonctions purement mathématiques.

Exemple : le rôle des états (suite et fin)

- Cette dualité est traitée différement selon les techniques formelles.
- Par exemple, dans la méthode B existe une notion d'*affectation simultanée* dont l'intérêt, en regroupant une suite d'affectations qui n'interfèrent pas, est de réduire le nombre d'états.
- Par ailleurs, les *langages fonctionnels* offrent une sémantique sans notion d'état, mais permettent aussi d'employer ceux-ci : le soin est laissé au programmeur de n'utiliser les états (on parle de *traits impératifs* en programmation, comme les boucles et les tableaux) que de façon périphérique.

Privilégier la spécification ou la vérification

- Une méthode formelle se bâtit à partir de deux ingrédients principaux : un langage de spécification et un système de vérification. Ces deux éléments sont inégalement développés selon les approches et les outils.
- Ainsi les systèmes d'assistance à la preuve de Boyer-Moore privilégient l'automatisation des démonstations au détriment de la facilité d'expression. À l'opposé, la conception de Z visait avant tout la capacité d'expression et a conduit à un langage difficile à outiller.
- Les approches plus récentes, comme HOL et Coq, tentent de conjuguer le meilleur des deux mondes : s'appuyant sur des logiques puissantes, elles viennent avec des outils qui, d'une part, aident à la construction des preuves et, d'autre part, permettent de vérifier ces dernières complètement avec une grande confiance.

Présentation des outils logiques

La logique mathématique s'est développée suivant plusieurs axes :

- la théorie des modèles,
- la théorie de la démonstration,
- la théorie des ensembles,
- la théorie des types,
- la théorie de la calculabilité.

L'importance de la logique dans le contexte du génie logiciel se résume à deux points :

- elle fournit un cadre pour exprimer de nombreuses notions ;
- elle se prête bien à la formalisation.

La logique en tant que cadre

Concernant le premier point :

- Les données manipulées par les programmes peuvent être décrites par des combinaisons d'ensembles élémentaires (entiers, caractères etc.) telles que des produits cartésiens (pour les `struct` du langage C) ou des unions (pour les `union` de C) par exemple.
- La théorie de la calculabilité nous apprend l'existence des limites, c'est-à-dire ici de spécifications irréalisables.
- La théorie des types conduit à des compilateurs qui garantissent la sûreté de la programmation (par le biais de la vérification et de l'inférence statique de types).

La logique pour la modélisation

Concernant le second point, il y a deux intérêts à formaliser :

- la rigueur des textes et des raisonnements est augmentée car on peut les ramener au moins en partie à des manipulations interprètables de symboles ;
- le travail induit peut être assisté ou, parfois, automatisable — dans tous les cas outillé.

La théorie des modèles

Il y a essentiellement deux façons de spécifier qui se complètent :

- représenter un système par ses propriétés,
- en donner un modèle constructif.

On parle parfois de *spécification par propriétés* ou *par modèles*. Les propriétés sont exprimées par des axiomes logiques et les modèles par des opérations ensemblistes. On parle dans le premier cas d'aspect syntaxique et dans le second cas d'aspect sémantique (car il décrit l'univers du discours lui-même). Par exemple, des modèles satisfaisant l'énoncé $\forall x.\exists y.(y\, R\, x)$ sont $(\mathbb{N}, >)$ ou $(\mathbb{R}, <)$ ou $(\mathbb{N}, \text{« est multiple de »})$ etc.

La *conséquence sémantique* est fondamentale : un énoncé E est une conséquence sémantique des énoncés A, B, C etc. si *tout* modèle ayant les propriétés A, B, C etc. a également la propriété E.

La théorie de la démonstration

La relation de conséquence sémantique a l'inconvénient de devoir être vérifiée sur tous les modèles possibles et il peut y en avoir une infinité en général. On peut préférer alors une *relation de prouvabilité* : un énoncé E est dit prouvable (ou démontrable) à partir des énoncés A, B, C etc. si on peut construire une preuve formelle de E en n'utilisant que les hypothèses A, B, C etc. ainsi que les axiomes et les règles de la logique. L'énoncé E est dit *réfutable* si sa négation est démontrable.

Il faut néanmoins que les manipulations formelles conservent la sémantique des énoncés : c'est la *validité*. Si toute conséquence sémantique est démontrable alors la théorie est *complète*.

La théorie de la démonstration (suite et fin)

Indépendamment des liens entre les notions de conséquence sémantique et de prouvabilité, il est des questions intrinsèques à la seconde, comme par exemple : si E est démontrable, en existe-t-il une preuve ne comportant que des sous-formules de E ? Si oui, l'espace de recherche est considérablement réduit, ce qui a un impact sur l'assistance à la preuve.

L'étude des axiomes et des règles en tant que calculs formels (c'est-à-dire purement syntaxiques) ainsi que leurs relations avec la notion de conséquence sémantique constitue la théorie de la démonstration.

La théorie des ensembles de Cantor et le paradoxe de Russell

Il est utile de former un ensemble dont les éléments possèdent une propriété donnée — une telle définition est dite *par compréhension*. Si ce type de définition n'est pas contraint, on se retrouve dans le cadre de la première théorie des ensembles, dite naïve, qui s'est révélée contradictoire au début du XX^{e} siècle.

Une théorie est contradictoire si on peut prouver un énoncé et son contraire, donc n'importe quoi *in fine*.

Considérons le paradoxe de Russell. Formons par compréhension le curieux ensemble $R = \{x \mid \neg(x \in x)\}$. Si $R \in R$ alors R doit vérifier la propriété de ses éléments, soit $\neg(R \in R)$. Réciproquement, si $\neg(R \in R)$ alors R vérifie la propriété caractéristique des éléments de R, c.-à-d. $R \in R$. Donc, en posant la propriété $P = R \in R$, on a prouvé P et $\neg P$, ce qui est contradictoire.

La théorie axiomatique des ensembles de Zermelo-Fraenkel

Une façon de contourner l'écueil des paradoxes consiste à reformuler la théorie des ensembles de façon axiomatique (c.-à-d. avec des règles logiques) en introduisant une forme de *stratification des constructions*. Ainsi une propriété ne peut être définie arbitrairement en portant sur une propriété arbitraire, comme c'est le cas dans le paradoxe de Russell.

Cette théorie, proposée par Zermelo et Fraenkel, a permis une refondation des mathématiques sans que personne n'ait reproduit de paradoxe jusqu'à présent.

Les langages de spécification Z et B s'appuient sur cette théorie en y ajoutant un zeste de typage pour la rendre plus proche des préoccupations de l'informatique.

La théorie des types

Une autre voie pour éviter les paradoxes a été proposée par Russell lui-même : la théorie des types. Il s'agit aussi de stratifier les ensembles (et les propriétés) : au niveau 0 les individus ; au niveau 1 les ensembles d'individus (et les propriétés portant sur les individus) ; au niveau 2 les ensembles d'ensembles d'individus etc. Les types caractérisent ces couches. Ils sont associés aux individus et aux propriétés de base et contraignent la construction des couches supérieures.

En informatique, cette approche s'est révélée très fructueuse, comme le montrent de nombreux langages de programmation récents. Pour permettre la récursivité il faut une règle de typage plus permissive. L'inconvénient inévitable est alors de devoir autoriser des programmes qui ne terminent pas.

La théorie de la calculabilité

Un autre aspect fondamental est la question de savoir quelles sont les fonctions mathématiques qui sont calculables par des machines : c'est le sujet de la théorie de la calculabilité.

Dans les années 1940, plusieurs approches ont été proposées :

- les machines de Turing,
- le λ-calcul (Church),
- les fonctions partielles récursives (Gödel, Herbrand).

Chacune constitue un cadre où exprimer une notion de *procédure effective* (algorithme). La *thèse de Church* est que ces différents formalismes sont équivalents (calculent le même ensemble de fonctions mathématiques).

Les ensembles dénombrables

Est-ce que toutes les fonctions sont calculables ? Pour le dire, un détour par la théorie des langages formels s'avère utile.

Un ensemble est *dénombrable* s'il est fini ou peut être mis en bijection avec l'ensemble des entiers naturels $\mathbb{N}$. Le cardinal d'un ensemble infini dénombrable est noté $\aleph_0$. Par exemple

- L'ensemble des nombres pairs est dénombrable (la bijection triviale est $2n \mapsto n$). Généralement, tout sous-ensemble infini des naturels est dénombrable.
- L'ensemble des mots finis sur l'alphabet $\{a, b\}$ est dénombrable (la bijection se construit en classant les mots avec l'ordre lexicographique).
- Les nombres rationnels sont dénombrables.
- Les expressions régulières sont dénombrables.

La diagonalisation de Cantor

On peut se demander alors s'il existe des ensembles dont le cardinal est supérieur à $\aleph_0$, c.-à-d. non dénombrables. La réponse est oui : *l'ensemble des sous-ensembles d'un ensemble dénombrable n'est pas dénombrable.*

On prouve cela par un procédé appelé *diagonalisation*. Soit un ensemble dénombrable $A = \{a_0, a_1, \ldots\}$ et soit S l'ensemble de ses sous-ensembles.

Raisonnons par l'absurde et supposons que S est dénombrable et donc que $S = \{s_0, s_1, \ldots\}$.

La diagonalisation de Cantor (suite)

Construisons alors le tableau infini suivant :

	a_0	a_1	a_2	a_3	a_4	$\ldots$
s_0	$\times$	$\times$		$\times$		
s_1	$\times$	$\circ$		$\times$		
s_2		$\times$	$\times$		$\times$	
s_3	$\times$		$\times$	$\circ$		
s_4		$\times$		$\times$	$\circ$	
$\vdots$						$\ddots$

Il indique quels éléments de A appartiennent à quels éléments de S : une croix à la colonne i et la ligne j indique que $a_i \in s_j$.

La diagonalisation de Cantor (suite et fin)

Considérons l'ensemble $D = \{a_i \mid \neg(a_i \in s_i)\}$. Ses éléments sont marqués par un rond dans notre tableau (sur la diagonale). On a $D \subset A$, mais est-ce que D est bien listé parmi les s_j ?

Supposons qu'il existe k tel que $D = s_k$. Or $a_k \in D \Leftrightarrow \neg(a_k \in s_k)$, ce qui implique $D \neq s_k$, donc contredit l'hypothèse. Donc D n'est pas listé parmi les sous-ensembles de A alors qu'il est un sous-ensemble de A. Ceci contredit donc l'hypothèse que l'ensemble des sous-ensembles d'un ensemble dénombrable est dénombrable.

On note $2^{\aleph_0}$ le cardinal d'un ensemble non dénombrable, par analogie avec le fait que le cardinal de l'ensemble des sous-ensembles d'un ensemble fini de cardinal n est 2^n.

Problèmes indécidables

- L'ensemble des langages est l'ensemble des sous-ensembles d'un ensemble dénombrable (l'ensemble des mots), donc, d'après le théorème précédent, il n'est pas dénombrable.
- La solution à un problème correspond à un langage, donc le nombre de solutions (ou problèmes) est égal au nombre de langages et n'est donc pas dénombrable.
- Un algorithme est une chaîne finie, donc l'ensemble des algorithmes est dénombrable.

Il y a donc plus de solutions que de solutions calculables (par des algorithmes). On parle de *problèmes décidables* pour ceux dont la solution est calculable et de *problèmes indécidables* pour ceux dont la solution n'est pas calculable.

 Jean-François Monin.
Introduction aux méthodes formelles.
Hermès Science, 2000.
Seconde édition.

 Pierre Wolper.
Introduction à la calculabilité.
Dunod, 2001.
Seconde édition.