Autocode

Principes

Introduction : la crise du logiciel

Les méthodes et pratiques actuelles en termes de développement générique conduisent à du code dont on sait qu'il contient encore des erreurs, sans pouvoir toutes les détecter. Les techniques de test et de vérification les plus poussées à avoir été mises au point ne peuvent y remédier. Le coût consacré à ces tests très poussés, voire invasifs (approche par mutations), précédés d'une phase de relecture du code faisant une large moisson d'erreurs, représente trois à quatre fois celui de rédaction du code à tester, pour un résultat parvenant à la seule certitude d'une absence de certitudes quand à l'absence d'erreurs.

D'un autre côté, les méthodes formelles, avec le langage B et ses équivalents, parviennent à produire du code dont on a la certitude qu'il est conforme aux spécifications et dépourvu d'erreurs, par une preuve mathématique du résultat, dispensant ainsi des dispendieux tests. Malheureusement, la lourdeur de ces méthodes et les restrictions quand à leur usage confinent leur utilisation dans de très petits segments du marché logiciel.

Cette *fracture logicielle* est inacceptable.

Les clients de l'industrie logicielle tolèrent de moins en moins les *bugs* et leurs désagréments. Aucune autre industrie n'a bénéficié d'autant de tolérance, et celle-ci va rapidement disparaître, à mesure que l'informatique envahit à la fois le quotidien et les secteurs les plus critiques (transport, santé,...).

Les contraintes et exigences d'implantation elles-mêmes se diversifient et s'approfondissent : R.A.D., encombrement mémoire, temps réel et latences, consommation CPU et énergétique, facilité d'usage et vieillissement de la population,...

Une proposition

Premier étage de la fusée : la génération automatique du code

Dans l'art de la programmation contemporain, une [trop[1]] grande partie des erreurs est éradiquée par une revue de code soigneuse effectuée après l'écriture du code, et notamment les erreurs d'écriture, les inversions, les oublis. De nombreux défauts subsistent cependant, dont certains sont connus comme étant de mauvaises pratiques de programmation.

Toutes ces « mésécritures » seraient supprimées par la génération automatique du code par l'atelier de développement. Or plusieurs outils tendent à faire ce codage automatique, notamment à partir des diagrammes UML. Ils ne dressent que le squelette des classes, mais le passage à une rédaction complète serait un grand bond en avant pour notre industrie logicielle.

En laissant de côté la preuve mathématique, les méthodes formelles procèdent par rédaction de spécificités détaillées, puis par raffinements *mécaniques*[2] de la proposition de solution correspondante, jusqu'à obtenir au final une implantation conforme aux spécifications de départ. Il est ainsi possible d'envisager un processus de génération en plusieurs étapes, avec transcription en code intermédiaire, puis obtention d'un code final réalisant une implantation optimisée pour un contexte particulier (embarqué, temps réel,...).

Nous exposons plus loin les contraintes et progrès à faire pour obtenir de tels outils.

Un intérêt majeur est que le coût de résorption d'une erreur augmente avec le degré d'avancement du projet où elle est découverte. Éviter les erreurs dès le début est quasi miraculeux au plan économique pour les éditeurs (d'autant plus qu'ils n'hésiteront pas à valoriser pécuniairement le plus haut degré de qualité fourni par l'Autocode).

Deuxième étage : mathématisation et preuve du code généré

A partir du moment où nous avons une spécification pouvant être ensuite codée automatiquement, il est possible ensuite d'examiner chacune des structures utilisées par le générateur de code et d'assurer la mathématisation de celles-ci.

Pour certaines de ces structures, nous pourrons prouver que, pour un contexte bien précisé, le code ainsi généré est sans défaut, à la façon des méthodes formelles. D'autres pourront sans doute être avantageusement remplacées par des combinaisons prouvables.

Les parties d'une application générée à partir de ces structures seront ainsi prouvées, ne nécessitant plus de tests (toujours incertains quand à l'absence de bogues).

Certains traitements resteront en dehors du champ d'application de l'Autocode, mais ces parties de code écrites à la main seront connues comme peu fiables et pourront être relues et testées comme actuellement. Il appartiendra à l'éditeur et à son client de choisir les parties devant être prouvées, générées, et manuelles dans le logiciel à fournir. Au fil du temps et des progrès des sciences de l'information, le champ des exclus devrait se réduire comme peau de chagrin.

---

1 la facilité avec laquelle un relecteur supprime une montagne d'erreurs est symptomatique de la non fiabilité d'opérateurs humains en matière de codage.

2 au sens d'une application mécanique de principes et formules mathématiques.

Autocode première couche : code sans erreur

L'existant

Typage fort

Les derniers langages orientés objet opèrent un strict contrôle de type à la compilation, vérifiant la bonne adéquation entre le paramètre demandé par l'opération et l'objet passé par l'appelant.

Les ateliers de développement eux-mêmes procèdent à ce contrôle de type via l'introspection et la complétion. Ils savent aujourd'hui souligner les erreurs de type (méthodes inexistantes pour le type, types non conformes aux paramètres,...).

Génération de code

Nous pouvons considérer les compilateurs comme des outils prenant en entrée une spécification détaillée (c'est le [pseudo]code en langage de dernière génération), et la transcrivant en code d'implantation (code machine, bytecode, assembleur).

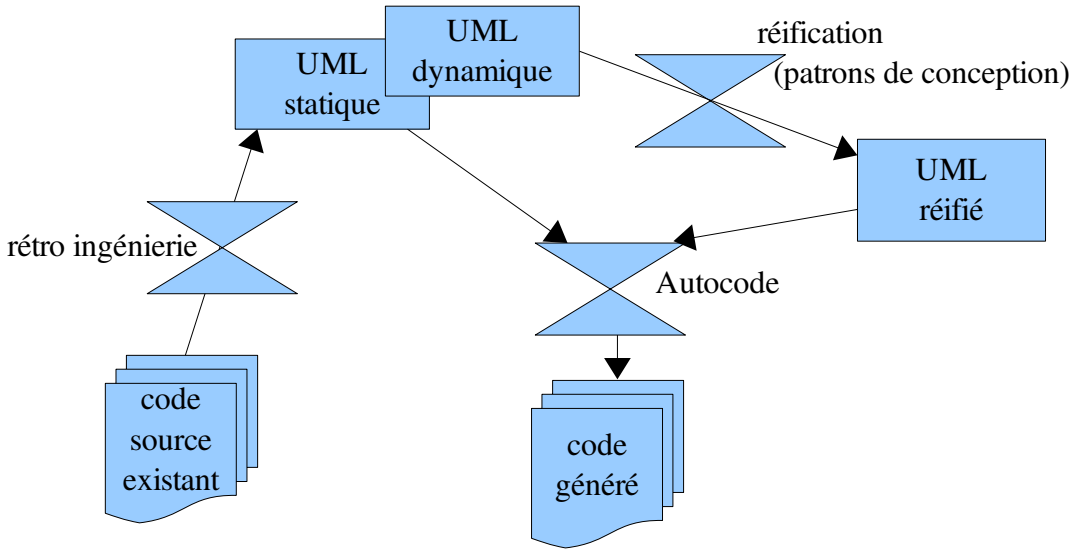
Les ateliers intégrés de développement disposent de plus en plus souvent d'outils permettant de spécifier visuellement les fonctions désirées et de générer le code correspondant aux structures ainsi définies (Eclipse + Omondo => UML vers Java).

Patrons de conception

Depuis C. Alexander et le GoF, nous disposons de structures génériques résolvant les difficultés les plus classiques du domaine informatique.

Notamment, certains de ces patrons (State, Command) permettent de transcrire les diagrammes dynamiques en structures statiques, par une réification.

Nous avons vu précédemment que des outils actuels permettent d'écrire le code correspondant.



Savoir-faire

Les revues de code sont très performantes en matière de chasse aux bogues grâce à la connaissance qu'ont les relecteurs des erreurs à ne pas commettre, ainsi que des erreurs les plus faciles à faire. Ce savoir peut être en grande partie capitalisé dans l'Autocode, au travers de structures à modéliser et d'assistants à développer.

Fracture homme / machine

Les capacités cognitives et la structure mentale des hommes diffèrent profondément de « celles » des machines. Ce qui est évident et immédiatement compréhensible pour l'un demande des efforts très complexes à l'autre pour arriver au même point.

Ainsi, la notation arithmétique infixe parenthésée est la plus accessible aux êtres humains, alors que son analyse est problématique pour les compilateurs. A l'inverse, la notation préfixée ou postfixée, étonnamment limpide pour une machine à pile, demandera un entraînement assez long aux humains pour qu'ils la manipulent aisément.

L'Autocode permet de départager les deux mondes sans ambiguïté, conservant un haut niveau d'efficacité aux deux parties.

Une formulation à définir

Besoins « clients »

Les spécifications doivent pouvoir être établies d'une façon qui rende leur rédaction accessible à des analystes bien formés, sans requérir pour autant un rigorisme lourd et coûteux en temps à passer et en expérience accumulée. Des systèmes de correction d'erreurs grossières, de questionnement aux embranchements, de levée des ambiguïtés,... permettraient d'assurer à leur utilisateur et à son client la conformité aux règles et restrictions d'usage de l'outil de spécification (voir ci-après).

Une des principales limitations du langage B est la nécessité de passer par une spécification mathématique suivie de transformations elles aussi mathématiques. Il est hautement préférable de spécifier via des abstractions accessibles à l'intuition commune (telles que l'Itérateur de Java, vu plus loin), branchées directement sur des concepts mathématiques purs. cela évitera aux êtres humains un apprentissage long et laborieux et aux machines une série de transformations difficiles à programmer (et même impossibles dans certains cas).

Pour quelles erreurs

Les différentes erreurs possibles en programmation semblent, a priori, rétives à toute tentative d'éradication. L'art de la programmation lui-même n'est guère propice à quelque optimisme quand à son automatisation.

Aussi, il va falloir recenser ces erreurs et, pour l'une ou l'autre, réaliser des études de normalisation afin de tenter d'obtenir un moyen d'empêcher son apparition dans le code.

De proche en proche, l'ajout de nouvelles erreurs à l'*index* des prohibitions permettra d'étendre le domaine d'application de cette technique.

Certaines erreurs ne pourront être « effacées » qu'après de nombreux progrès autour du problème considéré (logique, sémantique, mathématiques,...), laissant pendant longtemps une part

conséquente des logiciels dans le frustrant état actuel de l'art (à moins de restreindre le logiciel à ce qui peut être traité rigoureusement).

Ainsi, une opération prenant deux paramètres identiques est source possible d'erreurs par inversion des objets passés. Trois remèdes peuvent être envisagés :

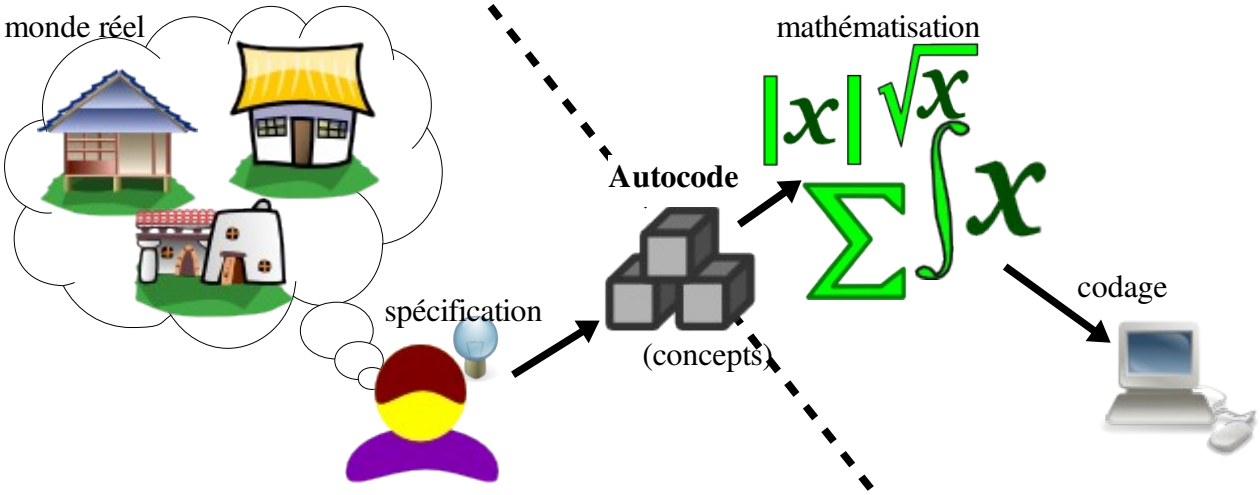
- `TracerPoint(int x, int y)` peut avantageusement être remplacée par `Tracer(Point p)`, avec `p.setX(int x)` et `p.setY(int y)` pour passer les valeurs nécessaires.
  On y gagne au passage le polymorphisme de `Tracer(Figure f)`.
- `Compresser(Fichier aCompresser, Fichier estCompresse)` applique un traitement sur le premier paramètre, en lien avec le second. Il vaut mieux alors utiliser une approche à la « `.equals()` ». L'opération est une possibilité d'agir sur le premier objet, le résultat concernant le second objet :
  `estCompresse = aCompresser.Compression().`
- Une troisième possibilité consiste à passer chacun des deux paramètres avant l'opération proprement dite. `ProduitCartésien(Liste a, Liste b)` est alors remplacé par :
  `setGauche(Liste a); setDroite(Liste b); ProduitCartesien();`

Il doit être possible à la fois d'abstraire ces trois concepts pour les présenter de façon humainement appropriable et de les mathématiser à l'usage de l'Autocode.

Une spécification de haut niveau

La spécification doit pouvoir être faite sans a priori sur l'implantation, ce qui suppose un moyen de décrire les détails des fonctionnalités avec un haut niveau d'abstraction, l'Autocode choisissant ensuite l'algorithme le mieux adapté compte-tenu des contraintes générales de l'application.

Un exemple nous est donné avec le parcours d'un ensemble d'objets : hier, le programmeur devait coder une liste, puis compter sa longueur, et enfin créer une boucle FOR avec un incrément. Aujourd'hui, il peut prendre une Collection et manipuler l'Itérateur de celle-ci avec une boucle ForEach. Aucun programmeur ne songerait à se plaindre de cette abstraction, et peu s'amusent à coder eux-même une fonctionnalité équivalente (à part quelques malheureux étudiants à qui on demande de réinventer la roue ;-)[3].



3 le mythe de Sisyphe a encore de beaux jours devant lui.
Hum... ce nom irait comme un gant à l'Autocode, puisqu'il a la charge de réaliser à chaque application les mêmes tâches pénibles et répétitives.

Fonctionnalités externes

Une bibliothèque pour les fonctionnalités non encore Autocodées

Nous avons vu que le parcours des ensembles d'objets pouvait être abstrait dans l'Autocode.

Cependant, tout ne sera pas abstractible immédiatement. Ainsi, les fonctionnalités particulières offertes par une liste doublement chaînées poseront peut-être des problème. Cela les cantonne dans une bibliothèque externe de l'Autocode, où les fonctionnalités restant nécessaires au développement seront préprogrammées de façon propre. Les utilisateurs de cette bibliothèque devront pouvoir vérifier et valider les choix d'implantation de leur rédacteur, ce qui implique la fourniture par celui-ci du code source de ces fonctionnalités.

Des bibliothèques de bas niveau transparentes

La plupart des langages modernes mettent à disposition des programmeurs des bibliothèques fournissant des structures élaborées leur évitant d'avoir à coder des fonctionnalités récurrentes d'une application à l'autre. Il s'agit à la fois d'un progrès, équivalent aux patrons de conception, et d'un affaiblissement, relativement à la question des erreurs. Seule la confiance dans la qualité du travail du fournisseur de ces bibliothèques assure de l'absence d'erreurs dans ces bibliothèques. Or nous avons vu que les tests les plus poussés ne peuvent offrir une telle garantie.

Pour tirer profit de l'approche Autocode, il s'agit cependant d'avoir des certitudes. C'est pourquoi l'Autocode produit du code (lisible par tout être humain) et non directement du langage machine (ce qu'il pourrait faire). Par ailleurs, le choix d'implantation fait lors de la génération de ce code permet une optimisation en fonction de contraintes fonctionnelles.

Aussi les fournisseurs de bibliothèques devront fournir soit des bibliothèques rédigées de haut niveau pour l'Autocode, permettant l'optimisation (et le choix du langage de bas niveau), soit le code source, permettant la relecture et les tests, à la façon des bibliothèques pour les schémas non encore abstraits précédemment évoqués.

Autocode deuxième couche : code prouvé

La première couche supprime les erreurs de codage, et avec elles la revue de code. Mais elle n'offre pas une certitude absolue quand à l'absence complète de défauts d'une forme ou d'une autre. Elle continue donc de nécessiter une batterie de tests lourde et coûteuse, à la différence des méthodes formelles (langage B).

Une démonstration à établir

Pour se débarrasser des tests ET de l'incertitude qu'ils laissent quand à la présence d'erreurs dans le code, sur au moins une portion du domaine logiciel, il convient de démontrer dans l'ordre les assertions relatives aux 4 niveaux considérés ci-après.

Il est vraisemblable que différentes manières de procéder à ces démonstrations induiront différents autocodes, correspondant à différents domaines d'application de l'industrie logicielle.

Niveau 1 : suppression (d'erreurs)

Il est possible de construire un générateur de code qui prenne en entrée une spécification d'une forme donnée et produise en sortie un code dépourvu d'erreurs d'un type donné

Niveau 2 : restriction (de domaine)

Une application logicielle restreinte à des spécifications compatibles avec ce générateur conduit à un ensemble de code sans erreurs.

Niveau 3 : composition (de structures)

Il est possible de construire des compositions d'autocodes qui n'introduisent aucune erreur.

Niveau 0 : exécution (contexte d')

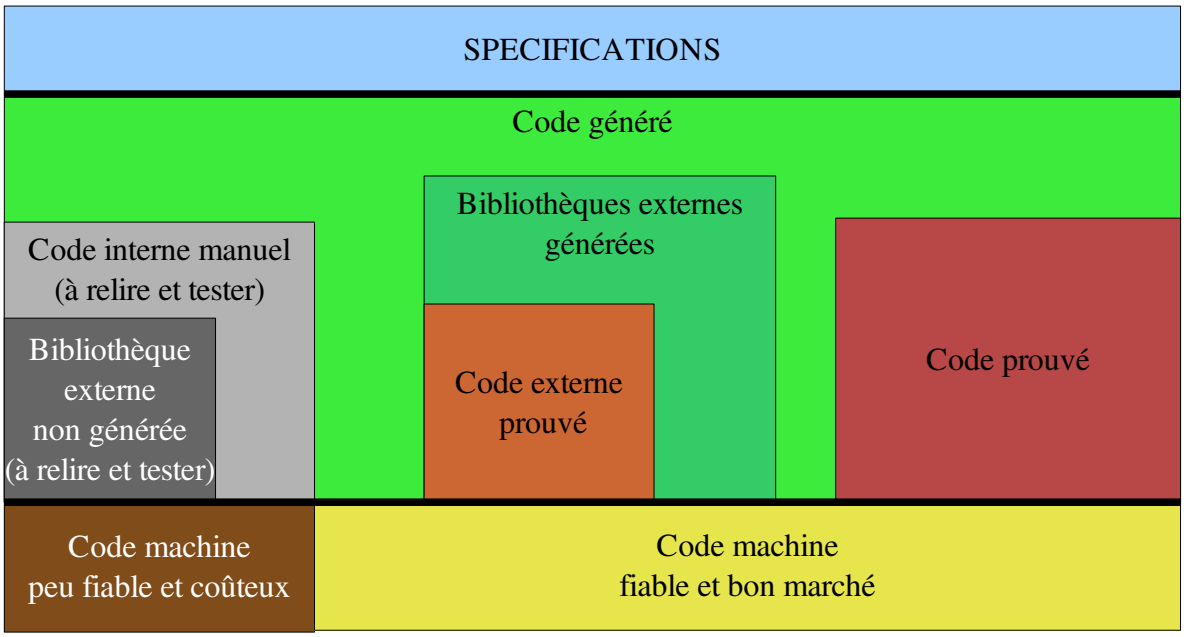
Il est possible de connaître et mettre en œuvre les restrictions d'environnement au sein desquelles l'exécution du code obtenu ne génère pas d'erreurs.

Mathématiques et programmation

Les boucles doivent pouvoir être modélisées par des suites décroissantes, pour assurer leur terminaison. Les premières seraient des suites arithmétiques, puis géométriques. Au fur et à mesure de leur preuve de bon déroulement et de terminaison, d'autres boucles, correspondant à des suites plus sophistiquées, pourraient être mises à disposition des informaticiens.

Résultat : une application plus solide

Répartition du code utilisé pour l'application



Remarques

Le premier étage seul ne suffit pas à répondre aux enjeux à venir de notre industrie. Aussi, le langage de spécification doit dès le départ être compatible avec son « 2e étage » : les structures proposées doivent être suffisamment précises et non ambiguës pour être prouvées, ou remplacées de façon transparente à l'utilisateur humain par un équivalent mathématisable.