Développement et validation de logiciels. Méthodes formelles

par **Patrick BELLOT**
Département informatique. École Nationale Supérieure des Télécommunications

Jean-Philippe COTTIN
Département informatique. École Nationale Supérieure des Télécommunications

et **Jean-François MONIN**
France Télécom, Centre National d'Études des Télécommunications, Lannion

Alors que les ingénieurs électroniciens savent concevoir des matériels qui fonctionnent et que les ingénieurs du bâtiment savent construire des ouvrages qui tiennent, l'ingénieur informaticien semble incapable de produire des applications répondant parfaitement aux besoins qu'elles doivent satisfaire. Quiconque a connu l'industrie de l'informatique sait que lorsqu'une application est produite, il y a un nombre très élevé d'allers et retours entre l'équipe de développement et les utilisateurs de l'application afin que celle-ci soit modifiée. Et même lorsque tout semble satisfaisant, on peut être certain que des problèmes surgiront encore. Les raisons souvent avancées sont le manque de précision ou l'incohérence dans l'expression des besoins des utilisateurs, ce que l'on appelle

les spécifications. Ces raisons peuvent être aussi la non-conformité de l'application par rapport aux spécifications ou bien encore de simples problèmes de fonctionnement de l'application.

La spécification en langage dit naturel permet au spécificateur d'utiliser les moyens qui lui semblent les plus appropriés à ce qu'il veut décrire : le français ou l'anglais, un exemple de comportement, une table, un programme, une référence, etc. Ces moyens sont, de plus, accessibles au plus grand nombre d'informaticiens. En revanche, ils peuvent favoriser l'apparition d'ambiguïtés dissimulées dans un verbiage excessif ou peu structuré. Ils ne permettent aucune analyse sérieuse de détection des incohérences ou des incomplétudes dans les spécifications. Ils ne permettent ni d'obtenir une application certifiée conforme aux spécifications, ni de vérifier a posteriori qu'une application est conforme à ces mêmes spécifications.

Les techniques formelles et leurs langages favorisent la maîtrise du cycle de vie du logiciel :

- *ils permettent une expression formelle des spécifications se prêtant à une interprétation précise, mathématique ;*
- *ils permettent une analyse qualitative (vérification de cohérence) et quantitative (vérification de complétude) des spécifications ;*
- *ils permettent, dans certains cas, la production semi-automatique de programmes certifiés conformes aux spécifications ;*
- *ils permettent aussi la preuve a posteriori que l'application est bien conforme aux spécifications.*

Spécifier formellement oblige en pratique à fouiller beaucoup plus profondément que ne le permettent les raisonnements ordinaires. L'expérience montre que la connaissance retirée d'un problème permet d'aborder le processus de réalisation avec beaucoup plus de confiance, car les descriptions sont très rigoureuses. Même dans un cadre où l'application produite ne sera pas confrontée formellement avec les spécifications originales, l'apport est fondamental. Il reste que la vérification formelle de la conformité d'une application à ses spécifications est parfois possible et figure l'élément concluant et fondamental des techniques formelles.

Les techniques formelles de développement de logiciel ont beaucoup progressé au cours de la décennie écoulée. Il est devenu possible de traiter certaines applications réelles de manière entièrement formelle. Cependant, le discours de ses promoteurs s'est assagi, tant sur la taille des applications que l'on peut prétendre traiter que sur le degré de confiance que l'on peut obtenir. Ces techniques sont donc appliquées spécifiquement à certaines applications qui n'ont aucun droit à l'erreur, telles que des logiciels de pilotage d'engins. Elles sont également utilisables, théoriquement, dans un cadre plus général pour minimiser le prix de revient d'une application en augmentant la fiabilité et en diminuant les coûts de maintenance et d'évolution de celle-ci.

1. Généralités

1.1 Bref historique

L'une des premières tentatives de rationalisation de la programmation fut l'introduction d'assertions dans le corps des programmes impératifs représentés par des organigrammes [11]. Ces assertions sont des formules logiques décrivant un état des éléments du programme. On distingue les préconditions qui doivent être vérifiées avant l'appel du programme, les post-conditions qui seront vérifiées après l'exécution de celui-ci et les invariants de boucle qui traduisent la rémanence d'une propriété pendant l'exécution d'une structure itérative. L'utilisation des assertions et des règles du système permet de démontrer formellement des propriétés du programme. Cette méthode fut partiellement automatisée pour des programmes de type ALGOL.

La méthode fut étendue et généralisée jusqu'à inclure des éléments de programme dans les assertions [17]. Dans le système de Hoare, les propriétés des programmes sont exprimées par des triplets $\{P\}\ S\{Q\}$ où S est un programme, P sa précondition et Q sa post-condition. On dispose de règles de la forme :

$$\frac{\{P \wedge B\}\, S\, \{Q\}}{\{P\}\ \text{while } B \text{ do } S\ \{P \wedge \neg B\}}$$

signifiant intuitivement que si S établit Q à partir de la précondition P et B, alors while B do S établit P et non B à partir de la précondition P. De telles règles sont utilisées pour démontrer la véracité des assertions figurant dans le programme. Elles furent même proposées [4] comme outil de définition de la sémantique d'un langage. La méthode de Hoare et ses descendants, par exemple la méthode des plus faibles conditions de Dijkstra [8], furent très longtemps les seuls outils permettant de prouver formellement, à la main ou de manière semi-automatique [15] [19] [29], les propriétés d'un programme. Il

est à remarquer que des langages récents comme EIFFEL [21] mais aussi C et C++ proposent d'inclure explicitement les assertions dans la syntaxe.

Une approche différente et plus ancienne trouve ses fondements dans LISP et les travaux de McCarthy [20] sur la **programmation fonctionnelle**. Dans ce style de programmation représenté par LISP ou d'autres langages plus rigoureux comme ML [22] [33], le programme est un objet mathématique qui apparaît comme sa propre spécification et est directement sujet à une étude formelle sans l'introduction d'un formalisme supplémentaire. On retrouve ces caractéristiques dans les systèmes de programmation en logique [3] [5] où les formules logiques tiennent lieu de programmes. Ces langages, sous leur forme pure, sont dits déclaratifs. En pratique, cet idéal doit faire face à des compromis pour des raisons d'efficacité ou lorsqu'il est nécessaire d'exprimer certains concepts, par exemple pour prendre en compte les changements subits par l'environnement dans lequel est plongé le programme considéré. Ces langages sont généralement réservés au maquettage d'applications ou au traitement de problèmes spécifiques nécessitant souplesse et dynamicité.

Les **techniques ensemblistes** sont représentées principalement par Z [32], B [1] et VDM [18]. Elles occupent une place de choix, car elles sont actuellement utilisées par l'industrie informatique et nous y reviendrons de manière plus détaillée (§ 4). Nous qualifions ces techniques d'ensemblistes, car les spécifications y sont énoncées au moyen de notations provenant de la théorie des ensembles et leurs fondements appartiennent à la tradition logique de la théorie axiomatique des ensembles. Des outils de preuves et de raffinage semi-automatique, intégrés dans des ateliers de génie logiciel (AGL), visent à permettre l'écriture de programmes certifiés corrects par rapport aux spécifications.

Les **techniques de spécification algébrique** [12] s'appuient sur la notion de type abstrait algébrique. Nous y reviendrons également (§ 5). Des développements récents ont abouti à des langages et des environnements proposant assistance au moyen d'outils d'édition, de modularisation et de preuves semi-automatiques.

Nous verrons également les **techniques de descriptions formelles** (FDT) incarnées par Estelle, LOTOS et SDL (cf. Normalisation).

Ces langages permettent de décrire des systèmes composés de processus parallèles communiquant par messages. LOTOS propose une vision algébrique, tandis qu'Estelle et SDL utilisent la notion d'automate et sont assez proches des langages de programmation de haut niveau usuels. SDL, en particulier, est largement utilisé dans le monde des télécommunications, car il dispose de bons outils d'édition et de génération de code.

Une tendance dont nous parlerons peu est celle de l'élévation du niveau des langages de programmation. La vision idéale est celle d'un langage de spécification confondu avec le langage de programmation. La spécification de l'application est alors compilée pour obtenir le programme de l'application ou interprétée pour en valider une exécution [30] [31].

Enfin, nous nous devons d'explorer ce qui peut être l'un des avenirs du développement de logiciel, c'est-à-dire l'**utilisation des théories logiques** et de leurs langages pour l'expression des spécifications ainsi que pour la génération automatique de programmes par extraction à partir de preuves d'existence [27] [10]. Les logiques utilisées sont alors non classiques : constructives, intuitionnistes, linéaires, etc. Elles relèvent d'une tradition logique différente de la théorie axiomatique des ensembles : la théorie des types.

1.2 Niveaux d'utilisation des méthodes formelles

On peut distinguer plusieurs niveaux d'implication des méthodes formelles dans le processus de développement d'une application. Il existe un niveau adapté à chaque cas de figure, prenant en compte les exigences de délais, de coûts, de qualité, de sécurité et l'état de la technologie. La classification suivante est empruntée à [28].

— **Niveau 0** : aucune utilisation d'une méthode formelle. Les documents sont exclusivement en langue naturelle utilisant éventuellement du pseudo-code et des diagrammes explicatifs. Leur utilisation est purement manuelle et la validation du logiciel obtenu est faite par des procédures extensives de tests. Il serait cependant hâtif de conclure qu'un tel développement est purement non formel : la programmation est bel et bien une formalisation. Mais les langages de programmation couramment utilisés se prêtent mal au raisonnement et à l'analyse mathématique.

— **Niveau 1** : utilisation de notations et de concepts formalisés, par exemple tables ou organigrammes. Le langage de spécification est plus rigoureux mais permet difficilement d'approcher les détails. Les preuves, quand il y en a, restent informelles. Ce niveau complète habilement une méthode classique de développement.

— **Niveau 2** : utilisation de notations formelles et d'outils semi-automatisés. Les spécifications sont rédigées dans un langage mathématique : théorie des ensembles, théories logiques, théorie des types, etc. Des outils informatisés permettent certaines facilités : aide à l'édition, vérification de consistance, gestion des dépendances, etc. Ce niveau est parfois atteint par de petits projets de développement dans un domaine très qualifié.

— **Niveau 3** : utilisation de notations formelles et d'un environnement d'outils détaillés et complets. Comme au niveau 2, les spécifications sont mathématiques et rigoureuses. L'utilisation d'outils plus ou moins automatiques d'assistance à la preuve sécurise l'ensemble du processus de développement du logiciel. Le logiciel est certifié conforme aux spécifications initiales. Tant en raison des limitations techniques actuelles de ces outils que pour des raisons évidentes de coût de développement, ce niveau de rigueur est réservé à de petites applications – éventuellement complexes – dans des domaines très spécifiques.

2. Méthodes formelles et cycle de vie du logiciel

Dans le cycle de vie d'un logiciel, les méthodes formelles apparaissent à différentes étapes et de plusieurs manières. Bien entendu, la rédaction des spécifications de l'application doit être réalisée dans un langage rigoureux ne donnant pas prise aux interprétations. Lorsque cette phase est terminée, plusieurs actions sont possibles :

— la validation des spécifications : par exemple, la vérification par évaluation sur un modèle (*model checking*), utilisée pour les protocoles, permet de valider un protocole en testant toutes les exécutions possibles et en y recherchant si telle propriété attendue est vérifiée ;

— l'étude de la cohérence des spécifications : par exemple, dans une description axiomatique il est important de savoir si les axiomes définis par l'utilisateur admettent un modèle non vide. Le système de vérification PVS, par exemple [30], possède des outils mécanisés qui vérifient certaines conditions de cohérence ;

— le prototypage des spécifications : si l'on considère PROLOG ou les langages fonctionnels de type LISP ou ML sous leurs formes pures comme des langages permettant d'écrire des spécifications exécutables, ils servent alors d'outils de prototypage des spécifications. Des systèmes tels que VDM ou B possèdent des interpréteurs permettant d'exécuter les spécifications.

Lors du développement proprement dit de l'application, l'utilisation de méthodes formelles a aussi son mot à dire :

— les méthodes de type Hoare ou Dijkstra permettent de démontrer manuellement que le code vérifie localement ses pré- et post-conditions. Celles-ci sont complétées par le calcul du raffinage de Morgan appliqué aux algorithmes [23] ou aux données [26] ;

— il existe différentes techniques de raffinage permettant de transformer par étapes la spécification, en général très abstraite, en un programme exécutable. Les étapes de ce raffinage qui sont prouvées en garantissent la sûreté. Une technique de raffinage est en général très liée au langage de spécification.

Lorsque le développement est terminé, l'utilisation d'une méthode formelle, Estelle ou VDM par exemple, permet la génération automatique ou semi-automatique de jeux de tests. Cela permet de valider *a posteriori*, sans la garantir totalement, la mise en œuvre qui a été faite des spécifications. On distingue les tests générés directement à partir des spécifications des tests obtenus à partir de comportements simulés.

Les apports énoncés ici se retrouvent également dans les évolutions futures de l'application spécifiée. Toute modification ou tout complément des spécifications pourra être validé, prototypé, raffiné, testé selon les mêmes méthodes. C'est donc une garantie de plus quant à l'évolutivité de l'application.

Il est bien évident que le coût de ces méthodes est élevé. Le calendrier est allongé par les étapes formelles qui ne sont pas directement liées au codage de l'application, par exemple les démonstrations de cohérence. Ces méthodes nécessitent, pour être applicables, des ateliers informatisés en règle générale très lourds. Actuellement, il n'existe pas encore de méthodes formelles permettant de sécuriser, de manière réaliste, le cycle dans son entier. Si nous prenons la technologie B de J-R. Abrial pour laquelle il existe de bons outils informatiques, et dont le développement est parmi les plus avancés, le développement d'une application raisonnable demande la démonstration d'une telle quantité de théorèmes que l'on finit toujours par être tenté de demander au système d'en accepter certains comme axiomes.

3. Bases mathématiques

On reconnaît trois axes mathématiques principaux qui servent de supports aux méthodes formelles : la logique du premier ordre, la théorie des ensembles et la théorie des types. La logique se rapporte au raisonnement tandis que théorie des ensembles et théorie des types sont les deux avatars du célèbre paradoxe de Russel.

3.1 Logique du premier ordre

La logique est en premier lieu un outil syntaxique pour énoncer des propriétés. La signification de ces énoncés logiques est définie par la notion d'interprétation et de modèle. En second lieu, la logique permet de faire des démonstrations rigoureuses, vérifiables et communicables.

3.1.1 S'exprimer et spécifier en logique

La logique du premier ordre n'est pas la plus puissante des logiques, mais elle est relativement simple et permet d'exprimer les propriétés de nombreux objets informatiques. Afin de désigner les objets de la spécification, la logique utilise des symboles d'individus tels que téléphone ou bien Caroline. La logique utilise également des symboles de fonctions comme numéro : numéro (x) servirait à désigner le numéro de téléphone d'une personne x. Enfin, elle utilise des symboles de prédicats servant à exprimer des propriétés des objets. Ainsi, on peut choisir que le prédicat est_abonné (x) signifie que la personne x est un abonné du téléphone.

L'utilisateur a donc toute latitude concernant le vocabulaire qui servira à exprimer les formules élémentaires. Celles-ci peuvent ensuite être combinées au moyen de connecteurs qui eux sont bien définis : $\wedge$ pour ET, $\vee$ pour OU, $\neg$ pour NON, $\Rightarrow$ pour «implique » ; $\forall$ pour « quel que soit », $\exists$ pour « il existe ».

3.1.2 Démontrer et vérifier en logique

Un intérêt majeur de la logique, qui complète son pouvoir d'expression non ambiguë, est la possibilité de faire des démonstrations rigoureuses et vérifiables. La logique est donc un langage composé des formules auquel on ajoute un ensemble d'axiomes et de règles d'inférence. Le tout est appelé un système formel. Un axiome est une formule considérée comme vraie *a priori*. Par exemple, certaines présentations de la logique admettent toutes les formules de la forme $A \Rightarrow (B \Rightarrow A)$ comme axiomes. Une règle d'inférence est une règle qui permet de déduire une formule, la conclusion, à partir d'autres formules que sont les prémisses. Par exemple, la règle dite *modus ponens* :

$$\frac{A \Rightarrow B \quad A}{B}$$

exprime que de $A \Rightarrow B$ et de A prouvés séparément on peut déduire B.

L'utilisation des axiomes et des règles d'inférence permet de construire un arbre dont :

— les feuilles sont soit des axiomes, soit des hypothèses ;

— chaque nœud correspond à l'application d'une règle d'inférence ;

— la racine est le théorème démontré.

Un tel arbre, appelé *arbre de preuve*, formalise la notion de démonstration.

La présentation de la logique du premier ordre sous la forme d'un système formel présente plusieurs avantages :

— on peut utiliser des démonstrateur semi-automatiques de théorèmes dont le résultat est garanti correct ;

— on peut utiliser des vérificateurs d'arbres de preuves entièrement automatiques.

3.1.3 Modèles

Les deux propriétés précédentes sont obtenues parce que le système formel, comme son nom l'indique, ne s'occupe que de la forme, donc de la syntaxe des objets qu'il manipule. Ce jeu formel est cependant loin d'être arbitraire. Il est possible de faire correspondre chaque élément syntaxique à un *modèle* mathématique relié à la réalité concrète que l'on veut représenter. Ainsi, dans l'exemple ci-dessus, Caroline correspond a un élément d'un ensemble de personnes, le rédacteur a vraisemblablement en tête une personne physique précise. Les prédicats sont interprétés par des fonctions vers la paire de valeurs de vérités $\mathcal{B} = \{\mathbf{v},\mathbf{f}\}$. Les connecteurs logiques comme $\wedge$ (ET), $\vee$ (OU) correspondent à des fonctions sur $\mathcal{B}$. Les axiomes ont la propriété d'être vrais – c'est-à-dire d'avoir la valeur **v** – dans tout modèle. Similairement, les règles, qui ont toujours pour forme :

$$\frac{P_1 \quad P_2 \ldots \quad P_n}{C}$$

où P_1, P_2... et P_n sont les prémisses et C la conclusion, sont justifiées par le fait que tout modèle dans lequel les P_i sont vrais satisfait également C. De proche en proche, on a ainsi la garantie que les théorèmes à la racine d'arbres de preuve sont toujours vrais.

3.2 Théorie des ensembles

Le paradoxe de Russel s'énonce ainsi : considérons l'ensemble de tous les ensembles qui n'appartiennent pas à eux-même. Soit :

$$R = \{x \text{ tel que } x \notin x\}$$

On peut, par un raisonnement simple, vérifier que $R \in R$ si et seulement si $R \notin R$. Il existe deux moyens de résoudre ce paradoxe apparu en théorie naïve des ensembles. Le premier moyen est d'admettre qu'il existe des prédicats tels que $x \notin x$ qui ne définissent pas des ensembles. En admettant ce postulat, on obtient la théorie des ensembles. Le deuxième moyen est d'admettre que l'on ne peut pas appliquer un prédicat à n'importe quels objets. On obtient alors la théorie des types, décrite plus en détail dans la suite.

3.2.1 S'exprimer et spécifier avec les ensembles

La théorie des ensembles fonde les langages de spécification ensemblistes tels que Z, B et VDM. Son principal atout réside dans la riche panoplie des opérations permettant de construire des ensembles complexes. La réunion notée $A \cup B$, la différence symétrique notée A/B, l'intersection notée $A \cap B$, le produit cartésien noté $A \times B$ vérifient nombre de propriétés algébriques : commutativité, associativité, idempotence, éléments neutres ou absorbants.

Pour illustrer à la fois la puissance de la théorie des ensembles et une technique d'utilisation, rappelons que la notion de fonction n'y est pas primitive, elle est manipulée à travers la notion de graphe, au sens d'ensemble de couples. Ainsi, on peut définir la fonction factorielle par l'ensemble des couples $\langle n, r \rangle$ tels que n est un entier positif et r est la factorielle de n. Pour commencer, on définit pour un ensemble la propriété de contenir le graphe de la fonction factorielle :

$$\text{contient\_fact}(E) = \langle 0, 1 \rangle \in E \wedge \forall n, r(\langle n, r \rangle \in E \Rightarrow \langle n + 1, n * r \rangle \in E)$$

Puis on définit le graphe de factorielle comme étant le plus petit ensemble vérifiant cette propriété :

$$\text{graphe\_factorielle} = \{c \mid \forall E(\text{contient\_fact}(E) \Rightarrow c \in E\}$$

3.2.2 Démontrer et vérifier avec des ensembles

La théorie des ensembles est suffisamment puissante pour permettre de représenter tout ce dont on peut avoir besoin, y compris les structures de données des programmes. Cependant, et on le voit dans l'exemple précédent, si les concepts sont relativement simples, l'expression peut faire appel à des techniques non triviales.

Sous sa forme axiomatisée, la théorie des ensembles est une théorie du premier ordre qui peut être étudiée en tant que système formel permettant des démonstrations automatiques ou semi-automatiques.

3.3 Notions sur les systèmes de types

Le typage est bien connu en informatique. C'est un garde-fou contre un certain nombre d'erreurs mais aussi un outil méthodologique.

En toute généralité, un type est simplement une expression formelle non interprétée et pouvant être rattachée aux éléments du langage informatique considéré. Un système de types est un système formel particulier permettant d'attribuer des types, de manière cohérente, aux éléments du langage. Par exemple, si l'on note $(A \to B)$ le type des programmes prenant un argument de type A et fournissant un résultat de type B, une règle du système formel sera :

$$\frac{f \text{ de type } A \to B \quad x \text{ de type } A}{f(x) \text{ de type } B}$$

La notion de type s'applique également aux langages de spécifications. Ainsi dans les langages de spécification algébrique, les symboles de types primitifs – les sortes – tels que nat et bool et les opérateurs comme $\times$ et $\to$ permettent de construire des types composés comme $\mathrm{nat} \times \mathrm{nat} \to \mathrm{bool}$.

Typage et spécification

Un système de types destiné a un compilateur vérifiant statiquement la conformité des programmes doit être décidable. Il ne peut apporter beaucoup d'information. Si une partie du contrôle de type – les indices de tableaux par exemple – est réalisé dynamiquement, on ne peut pas garantir l'absence de fautes à l'exécution. On verra plus loin qu'il existe également des systèmes de types très expressifs où l'information contenue dans un type est une véritable spécification logique. Mais alors la preuve de bon typage, c'est-à-dire le contrôle de types, doit être faite avec l'assistance du programmeur.

4. Méthodes ensemblistes

Les spécifications formelles utilisent des notations mathématiques pour décrire de façon précise les propriétés qu'un système d'information doit avoir. Des méthodes ensemblistes comme VDM ont été développées à partir des années 60, et cette approche continue avec l'apparition plus récente de la notation Z et de la méthode B.

4.1 Méthode VDM

VDM (*Vienna Development Method*) est plus fondée sur une vision dénotationnelle et opératoire – transformant ainsi les programmes en fonctions mathématiques – que sur une méthode basée sur la théorie axiomatique des ensembles, comme le sont **Z** ou **B**.

La méthode VDM utilise un langage appelé à l'origine Meta-IV et maintenant VDM-SL (VDM Specification Language). Elle donne la dynamique des systèmes informatiques spécifiés au moyen de prédicats de type **pré-** ou **post-**conditions.

VDM-SL permet une approche modulaire. Les données et les opérations peuvent être présentées comme des modules (passage des spécifications au programme) en utilisant des règles logiques sur les obligations de preuves qui sont systématiquement engendrées à partir de la théorie de VDM.

Comme en Z, on distingue les variables avant et après une opération par une décoration ; en VDM, la variable initiale est surmontée de « $\leftharpoonup$ ».

Le système logique employé par VDM comporte trois valeurs (vrai, faux, indéfini).

La valeur indéfinie est utilisée lorsqu'une fonction partielle apparaissant dans une expression logique est appliquée en dehors de son domaine de définition.

Malgré son nom, VDM est plus une notation qu'une méthode et elle ne contient pas d'outils d'aide à l'écriture de spécification ni de mécanisme de raffinage.

4.2 Méthode Z

Une spécification en Z est constituée d'ensembles que l'on utilise pour spécifier un système informatique et de schémas. Une spécification sera constituée d'un morcellement de schémas.

Les schémas sont utilisés pour décrire à la fois les aspects statiques et les aspects dynamiques d'un système. On entend par aspects statiques les états et les relations d'invariants entre les états du système et par aspects dynamiques les opérations qui sont possibles, les relations entre les entrées et les sorties et les changements d'états qui peuvent arriver.

Le nom du schéma est inscrit dans la première ligne du cadre qui l'entoure. Une ligne horizontale sépare la partie déclarative et la partie réservée aux prédicats. S'il y a plusieurs prédicats, on considère leur conjonction :

Nom_du_schema

Déclarations des variables

Partie prédicat

On peut également noter un schéma sous forme textuelle :

<Nom_du_schéma>==[*Declarations_des_variables*|*Partie_predicat*]

On y reconnaît tout simplement la définition d'un ensemble par compréhension. Les données d'un système sont munies d'un type. Ces types de données ne sont pas interprétés par une représentation informatique, mais par des ensembles construits au moyen d'ensembles prédéfinis (comme celui des entiers relatifs) et des opérateurs ensemblistes usuels (union, produit cartésien, etc.).

Les opérations sont spécifiées par un schéma avec les variables **avant l'opération** et les variables **après l'opération**. On distingue d'ailleurs les variables après l'opération par une apostrophe.

Incr

$x, x' : \mathbb{N}$

$x' = x + 1$

Une spécification Z utilise deux catégories de symboles :

— les symboles de la logique des prédicats $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\forall$, … ;

— les symboles de la théorie des ensembles, ceux de base ($\cup$, $\cap$ mais aussi le cardinal avec #, l'ensemble des parties finies d'un ensemble avec $\mathbb{F}$ …) ou les symboles fonctionnels et relationnels ($\leftrightarrow$, $\rightarrow$, des distinctions sont faites entre les fonctions injectives, surjectives, partielles ou totales [32]).

Plusieurs outils d'aide à la spécification et à sa vérification existent en Z. On distingue plus particulièrement fuzz développé à l'initiative de M. Spivey. Cet ensemble d'outils permet une aide à l'écriture d'une spécification et une vérification de types.

4.3 Méthode B

La technologie B est issue de travaux menés par J-R. Abrial dans les années 80, après ses travaux sur Z. Elle regroupe :

— un langage de spécification AMN (Abstract Machine Notation) assorti d'un système de preuves ;

— une technique de raffinage et de codage et les systèmes de preuves correspondants.

Comme Z, B repose sur la théorie des ensembles et la logique des prédicats du premier ordre. Contrairement à Z, B a une coloration développement dans la façon de spécifier les opérations : on ne les spécifie pas en terme de pré- et post-conditions, mais au moyen de *substitutions généralisées*. Ce mécanisme peut s'interpréter comme une extension de l'affectation telle qu'elle existe dans les langages impératifs, et que nous allons exposer dans la suite. Enfin, B offre une notation homogène pour la spécification et le développement.

4.3.1 Substitutions simples et substitutions généralisées

Reprenons l'exemple du système comportant juste une variable entière, et sur lequel on veut spécifier une opération d'incrémentation. En B, on n'utilisera que la variable x pour spécifier l'opération : $x := x + 1$. Il s'agit là d'une *substitution*.

Supposons que l'on veuille contraindre les valeurs prises par x en leur imposant d'être inférieures à 10. Intuitivement, pour établir la propriété $x < 10$ après avoir remplacé x par $x + 1$, il faut être parti d'une valeur inférieure à 9. De manière calculatoire, on obtient $x < 9$ en substituant $x + 1$ à x dans l'expression $x + 1 < 10$.

L'AMN comporte des opérateurs permettant de combiner des substitutions simples, ce qui donne un langage inspiré des commandes gardées de Dijkstra [8], travaillant sur des structures de données construites avec les opérateurs ensemblistes.

4.3.2 Processus de certification de B

Les machines abstraites utilisées pour la spécification utilisent des constructions non déterministes et toute la puissance du langage des ensembles. En revanche, les constructions algorithmiques (séquence, boucles) sont interdites. Au fur et à mesure des étapes de raffinage, les structures de données ensemblistes sont progressivement remplacées par des structures proches de celles des langages de programmation, le non-déterminisme est levé et l'on introduit des substitutions généralisées analogues à la séquence et aux boucles.

Toutes ces étapes sont soumises à des preuves de conservation des invariants et de conformité des machines raffinées par rapport aux machines plus abstraites. Elles sont supportées par des outils de vérification de syntaxe et de type pour l'AMN, un générateur d'obligation de preuves, un démonstrateur automatique qui utilise la logique classique [9] et depuis peu ces outils ont évolué [24] avec la traduction d'une implémentation en AMN en code C, des librairies de spécifications et la génération d'implémentations orientées objet.

5. Types abstraits algébriques

Les techniques de spécification algébrique bénéficient d'un corpus de résultats scientifiques bien plus développé que dans les méthodes ensemblistes. Il semble pourtant qu'elles restent relativement confinées au monde académique, faute peut-être d'avoir donné naissance à un ou deux langages suffisamment représentatifs. Il n'est cependant pas possible de les ignorer, car en sont issus un certain nombre de concepts qui irriguent les autres techniques de spécification et même d'autres domaines de l'informatique.

D'une manière générale, un type abstrait décrit une structure de données en masquant sa réalisation. Prenons l'exemple simple d'un arbre binaire : pour qui veut utiliser un arbre binaire, il importe de savoir en créer un, en assembler deux ou inversement considérer une branche, etc. ; il lui importe également de savoir de quelle forme d'arbre il s'agit : les informations sont-elles sur les feuilles ou sur les nœuds ? En revanche les choix de représentation qui peuvent mettre en œuvre des pointeurs, une arithmétique d'indexation dans un tableau, ou d'autres solutions ne le concernent pas.

On s'attache donc dans cette approche à décrire aussi précisément que possible l'*interface* de la structure de données. On a tout d'abord besoin de désigner les opérations qui la manipulent. Par exemple,

pour un arbre binaire dont les feuilles sont décorées par un entier, on peut considérer les opérations cbin qui construit un arbre à partir de deux sous-arbres, cfeu qui construit une feuille, sarbg (respectivement sarbd) qui extrait le sous-arbre gauche (respectivement droit), prof qui donne la profondeur, équil qui indique si l'arbre est équilibré, et ainsi de suite.

Bien sûr ce n'est pas suffisant : pour chaque opération il faut en indiquer la nature, on dira : la *sorte*, des objets manipulés, ici arbres, entiers ou valeurs booléennes. On les notera ici arb, ent et bool. En spécifications algébriques, les opérations que l'on considère sont des fonctions sans effet de bord. Ainsi toute expression formée au moyen de celles-ci correspond à une *valeur*, par exemple cbin (cfeu (3), cfeu (1)) représente un arbre ayant deux feuilles décorées par 3 et 1.

La déclaration des sortes des arguments et du résultat se formule au moyen d'une *signature*. Par exemple $\text{ent} \to \text{arb}$ est la signature d'une fonction prenant un entier en argument et rendant un arbre, $\text{arb} \times \text{arb} \to \text{arb}$ est celle d'une fonction prenant deux arbres en arguments et rendant un arbre.

Les opérations ci-dessus seraient déclarées ainsi :

$$\begin{aligned} \text{cfeu} &: \text{ent} \to \text{arb} \\ \text{cbin} &: \text{arb} \times \text{arb} \to \text{arb} \\ \text{sarbg} &: \text{arb} \to \text{arb} \\ \text{sarbd} &: \text{arb} \to \text{arb} \\ \text{prof} &: \text{arb} \to \text{ent} \\ \text{équil} &: \text{arb} \to \text{bool} \end{aligned}$$

Ce n'est toujours pas suffisant : rien n'est encore dit sur la signification de ces opérations, en dehors des commentaires ci-dessus. Pour cela on introduit des *axiomes*. Ceux-ci permettent de caractériser les arbres et leurs opérations, de même qu'en mathématiques on peut caractériser les groupes au moyen de trois axiomes. Par exemple, on peut énoncer qu'un arbre est équilibré si, étant formé de deux sous-arbres, ceux-ci sont eux-même équilibrés et que la distance entre leurs profondeurs ne dépasse pas 1 :

$$\forall a,\ b\ \ \text{équil}(a) = \text{true} \wedge \text{équil}(b) = \text{true} \wedge |\text{prof}(a) - \text{prof}(b)| \leqslant 1$$
$$\to \text{équil}(\text{cbin}(a,b)) = \text{true}$$

Les fonctions sarbg, sarbd et prof sont déterminées par les axiomes suivants :

$$\begin{aligned} &\forall a,b \quad \text{sarbg}(\text{cbin}(a,b)) = a \\ &\forall a,b \quad \text{sarbd}(\text{cbin}(a,b)) = b \\ &\forall n \quad\ \ \text{prof}(\text{cfeu}(n)) = 1 \\ &\forall a,b \quad \text{prof}(\text{cbin}(a,b)) = 1 + \max(\text{prof}(a), \text{prof}(b)) \end{aligned}$$

La sémantique d'une telle spécification est donnée par un modèle des axiomes, au sens défini dans le paragraphe 3.1. Un exemple de propriété aisément vérifiable est $\forall a$ prof (sarbg(a)) < prof(a). Une technique de preuve particulièrement développée et adaptée à ce contexte est la réécriture. Très grossièrement, on oriente les équations, et l'on substitue itérativement des membres gauches par les membres droits correspondants dans le but à démontrer jusqu'à obtenir une instance d'axiome. Il faut assurer que ce processus converge.

6. Domaine des protocoles

Les langages formels de description de protocoles qui rencontrent le plus de succès empruntent beaucoup moins aux notions de logique ou d'algèbre qu'à celles de l'informatique traditionnelle. Sur les trois qui sont normalisés à l'UIT et à l'ISO, un seulement, LOTOS, est d'origine algébrique, les deux autres, Estelle et LDS (SDL en anglais), sont basés sur la notion d'automates communicants et sont en fait assez proches de langages de programmation – et assez proches l'un de l'autre. Ce sont ces deux derniers qui sont les plus employés, LDS est notamment assez soutenu dans le monde des télécommunications.

Pour les besoins de la vérification, on introduit généralement un deuxième langage afin d'exprimer les séquences de comportement attendues. Le plus souvent il s'agit d'une logique temporelle, la vérification se faisant par une technique d'évaluation sur un modèle (*model checking*).

Pour illustrer cela, nous présentons sommairement Estelle, qui est plus simple que LDS, puis une version traditionnelle de la logique temporelle linéaire. Il existe des logiques temporelles plus riches et plus subtiles, en dehors des objectifs de cet article.

6.1 Estelle

Estelle permet de représenter l'architecture d'un système communicant et le comportement des éléments qui le composent. L'architecture est décrite au moyen de blocs appelés *modules* reliés par des canaux de communication. D'un bloc, on ne voit tout d'abord que l'interface, constituée de *points d'interaction*, qui sont déclarés avec un nom et une description des messages qui y passent. Ce mécanisme permet de typer les communications et donc d'effectuer un premier niveau de contrôle de cohérence. Les communications ne sont pas bloquantes pour l'émetteur : chaque point d'interaction est muni d'une file d'attente où sont stockés les messages arrivés.

À l'intérieur d'un bloc, on trouve soit une nouvelle structuration en sous-blocs, soit un automate.

La notion d'automate est élémentaire : on se donne simplement un ensemble d'états muni d'un état initial et un ensemble de transitions entre états. Chaque transition est définie par son état de départ, son état d'arrivée, un message optionnel en entrée et une séquence optionnelle de messages émis. Dans les cas très simples on peut se contenter d'énumérer les états. Mais dans le cas général cela est insuffisant. Par exemple un message reçu peut comporter un paramètre que l'on souhaite conserver pour traitement ultérieur. Pour permettre cela, l'état en Estelle est structuré en un vecteur de composantes. En pratique, les types des paramètres des messages sont des types Pascal sans pointeurs, l'état est défini par des déclarations de variables Pascal, l'état de départ d'une transition est défini par une expression booléenne portant sur ces variables et l'état d'arrivée est décrit par un corps de procédure Pascal.

Pour fixer les idées, la figure 1 donne un extrait d'un petit protocole de synchronisation d'horloges dû à G. Roucairol, dans lequel deux stations munies d'une horloge locale Hloc s'envoient de temps à autre leur date courante *via* un médium non fiable non représenté ici. Chaque station communique avec le médium *via* les points d'interaction HE et HS. Le protocole garantit que la distance entre les valeurs de Hloc des deux stations ne dépasse pas la constante DELTA.

```
channel canal_hor; date (v:integer); end;
module station; ip HE,HS : canal_hor; end;
body algo for station;
   var Hloc,HdistD : integer;
   initialize begin Hloc:=0; HdistD:=DELTA end;
   trans provided Hloc < HdistD begin Hloc:=Hloc+1 end;
   trans begin output HS.date(Hloc) end;
   trans when HE.date(v) begin HdistD:=max(HdistD,v+DELTA) end;
end;
```

Figure 1 – Un protocole de synchronisation en Estelle

6.2 Logique temporelle

La propriété attendue du protocole de la figure 1 s'exprime très simplement en logique temporelle : $\Box(|\mathrm{Hloc}_1 - \mathrm{Hloc}_2| \leqslant \mathrm{DELTA})$. L'opérateur $\Box$ signifie que la proposition à laquelle il est appliqué reste toujours vraie durant l'exécution. D'une manière générale $\Box$ permet d'exprimer des **propriétés de sûreté**, signifiant que rien de mauvais ne se produira... mais cela n'interdit pas les situations de blocage où plus rien ne se passe. L'opérateur dual $\Diamond$ permet d'exprimer des **propriétés de vivacité** signifiant que quelque chose de bon se produira. Dans notre exemple $\forall n \in \mathbb{N} \Diamond (\mathrm{Hloc}_1 > n)$ indique que la date de la station 1 n'est jamais arrêtée dans sa progression. Remarquons que cette propriété n'est pas vérifiée si l'automate choisit systématiquement la seconde transition de préférence à la première, si le médium se met à perdre tous les messages ou seulement, plus subtilement, tous les messages de la forme date (k) pour un k fixé. Intuitivement, on peut considérer que de tels comportements sont de probabilité nulle pour un médium réel – pas de génie de Maxwell – et il est possible de les interdire pour un automate dont on contrôle la réalisation. D'une manière générale, on considère qu'une action qui est possible infiniment souvent finit par s'effectuer, ce qui se formalise sous le nom d'équité. L'équité est souvent indispensable pour assurer une propriété de vivacité.

Les opérateurs $\Box$ et $\Diamond$, appelés opérateurs modaux, peuvent être combinés à loisir soit entre eux, soit avec les connecteurs logiques traditionnels, ce qui permet de spécifier facilement bon nombre de propriétés. Ainsi, pour exprimer qu'une situation caractérisée par la proposition P évolue nécessairement vers une situation caractérisée par la proposition Q, on introduit souvent l'opérateur binaire $\rightsquigarrow$ défini par $P \rightsquigarrow Q = \Box(P \Rightarrow \Diamond Q)$. Une autre manière de formaliser la progression de Hloc_1 est donc :

$$\forall n \in \mathbb{N}\ (\mathrm{Hloc}_1 = n) \rightsquigarrow (\mathrm{Hloc}_1 = n + 1)$$

6.2.1 Modèles de Kripke

Quelle signification accorder à des formules de logique temporelle ? La réponse est guidée par une idée intuitive simple : une proposition temporelle P peut être vue comme un prédicat $P(\_)$ sur un argument implicite, le temps. Dans le cas de la logique temporelle dite linéaire, le paramètre temps prend ses valeurs sur un espace totalement ordonné comme $\mathbb{N}$, P s'interprète comme $P(t_0)$, la valeur de P à l'instant courant t_0, $\Box P$ s'interprète comme $\forall t \geqslant t_0 \cdot P(t)$, $\Diamond P$ s'interprète comme $\exists t \geqslant t_0 \cdot P(t)$, $\Box(P \Rightarrow \Diamond Q)$ s'interprète comme $\forall t \geqslant t_0 \cdot P(t) \Rightarrow \exists t' \geqslant t \cdot Q(t')$ et ainsi de suite.

Plutôt que passer par une traduction dans le calcul des prédicats, on préfère adapter directement la notion de modèle. Rappelons que, dans le calcul des propositions, la sémantique d'une formule est calculée à partir d'une valuation des atomes vers {**v**,**f**}. En logique temporelle, on considère ces valuations comme des mondes, et un modèle n'est plus constitué d'un seul monde, mais d'un ensemble $\mathcal{M}$ de mondes reliés entre eux par une relation d'accessibilité réflexive et transitive notée $\leqslant$ et muni d'un monde initial m_0. Intuitivement chaque monde m de $\mathcal{M}$ représente un état possible du système complet, et $m \leqslant m'$ exprime que ce système peut évoluer de l'état m vers l'état m'. La sémantique des connecteurs logiques usuels se définit alors comme d'ordinaire, à ceci près qu'elle est relative à un monde de $\mathcal{M}$ et non à $\mathcal{M}$ tout entier. Par exemple $P \wedge Q$ est valide en m si et seulement si P et Q sont valides en m. Finalement, comme on peut s'y attendre, $\Box P$ est valide sur m si P est valide sur tout m' de $\mathcal{M}$ tel que $m \leqslant m'$ et $\Diamond P$ est valide sur m si P est valide sur un m' de $\mathcal{M}$ tel que $m \leqslant m'$. Enfin $\mathcal{M}$ est un modèle de P si et seulement si P est valide en m_0. Cette notion de modèle est due à Kripke.

6.2.2 Vérification par évaluation sur un modèle

Dès que l'on est muni de ces notions, il est facile d'expliquer celle de vérification par évaluation sur un modèle (*model checking*) : il s'agit simplement de déterminer, étant donné un modèle candidat $\langle \mathcal{M}, \leqslant \rangle$ et une propriété P qui est une formule de logique temporelle, si $\langle \mathcal{M}, \leqslant \rangle$ est un modèle de P. Dans le cas des protocoles, $\langle \mathcal{M}, \leqslant \rangle$ est le graphe d'états globaux construit à partir d'une description donnée par exemple en Estelle. En pratique, on commence généralement par calculer ce graphe, ce qui nécessite qu'il soit fini. En particulier, cela interdit d'utiliser des variables entières ainsi que des piles ou des files non bornées. Que valent ces restrictions ? À première vue, on peut estimer que, quelles que soient les qualités conceptuelles d'un langage de spécification, la réalisation du protocole sera limitée en mémoire ; il serait donc inutile d'employer des types de données trop généraux. Mais c'est oublier le caractère exponentiel du problème : il suffit d'introduire une variable codée sur un octet dans deux entités d'un protocole pour que le nombre d'états globaux soit multiplié par 32 000. En pratique, on est donc de toute façon condamné à ne traiter qu'une modélisation édulcorée, correspondant à des réalisations bien plus petites que ce que les limitations en mémoire autorisent.

La technologie courante permet de traiter des systèmes de plus de 10^{20} états, grâce à des algorithmes extrêmement sophistiqués. Cela correspond à des raisonnements combinatoires totalement hors de portée de l'homme et est largement suffisant pour attaquer des portions de systèmes réels. La vérification par évaluation sur un modèle convient particulièrement lorsque la complexité est placée au niveau du contrôle plutôt qu'à celui des données. L'un des grands intérêts de cette approche réside dans son automatisation : elle ne requiert rien d'autre de l'utilisateur que le minimum, c'est-à-dire la description d'un protocole et la formulation d'une propriété.

6.2.3 Démonstrations en logique temporelle

On aura remarqué au début du paragraphe 6.2.1 que la logique temporelle n'est pas plus expressive que la logique des prédicats ; mais elle offre une notation plus concise et mieux adaptée au raisonnement temporel.

Il est possible d'effectuer des démonstrations formelles en logique temporelle, en suivant par exemple une approche axiomatique (figure 2).

$$\Box(A \Rightarrow B) \Rightarrow (\Box A \Rightarrow \Box B)$$
$$\Box A \Rightarrow A$$
$$\Box A \Rightarrow \Box\Box A$$
$$\Diamond A \overset{\text{déf}}{=} \neg\Box\neg A$$

Figure 2 – Axiomes de la logique temporelle (système S4)

6.3 Approche synchrone

Si l'on cherche à raisonner sur un système structuré au moyen de modules communicant par messages asynchrones, on est amené à considérer tous les entrelacements d'événements ; cela est l'une des causes importantes d'explosion combinatoire du nombre de situations à prendre en compte. Il s'agit là d'une complication bien inutile lorsque l'on veut décrire un système centralisé. Dans ce contexte, on a avantage à utiliser un modèle synchrone, tel qu'on le trouve dans les langages Esterel (impératif) [14] et Lustre [25] (à flots de données). Ces langages sont adaptés à la description de systèmes réactifs et font l'hypothèse potentiellement paradoxale, dite « de synchronicité », que le temps de calcul d'une réponse est nul. Ainsi les sorties sont synchrones aux entrées. Cette hypothèse s'interprète de deux façons :

— si l'on considère un système réagissant à un environnement non maîtrisé, cela revient à supposer que le temps de réaction du système est plus petit que le temps qui sépare deux sollicitations de l'environnement ;

— si l'on considère les sous-systèmes d'un système structuré en modules, certaines techniques très astucieuses de compilation permettent d'annuler effectivement le temps de réaction d'un module à une sollicitation provenant d'un autre module.

La combinaison de l'hypothèse de synchronicité avec quelques autres choix de conception limite grandement l'explosion combinatoire des situations possibles, ce qui facilite l'analyse automatique.

7. Logiques d'ordre supérieur, logiques constructives

Lorsqu'on veut exprimer des spécifications et raisonner sur leurs propriétés, on peut être amené à introduire des fonctions mathématiques dont la complexité logique est arbitrairement grande. Cela est en particulier le cas si l'on veut exprimer des mécanismes généraux de manière uniforme.

7.1 Ordre supérieur

Prenons l'exemple même simple de la composition de deux fonctions. Chacun sait que $(g \circ f)(x) = g(f(x))$. La logique du premier ordre ne permet pas d'exprimer des assertions telles que :

$$\forall f \, \forall g \, \forall x \;\; (g \circ f)(x) = g(f(x))$$
$$\forall f \;\; f \circ \mathrm{Id} = f$$
$$\forall f \, \forall g \, \forall h \;\; h \circ (g \circ f) = (h \circ g) \circ f$$

En effet, ces assertions mettent en jeu des quantifications sur des fonctions.

De la même façon, la logique du premier ordre ne permet pas de quantifier sur des prédicats, ce qui permettrait d'énoncer le principe de récurrence en un seul axiome, ou une propriété d'hérédité :

$$\forall P \;\; [P(0) \wedge \forall n \, P(n) \Rightarrow P(n+1)] \Rightarrow \forall x \, P(x)$$
$$\forall P \;\; \text{héréditaire}(P) \Rightarrow [\forall x \forall y \, P(x) \wedge \text{descend}(x, y) \Rightarrow P(y)]$$

Observons que $\circ$ prend en argument deux fonctions et rend une fonction, et que héréditaire est un prédicat portant sur des prédicats. Cela est extrêmement intéressant du point de vue de la puissance d'expression, voire souvent indispensable si l'on veut exprimer simplement des principes suffisamment généraux pour être réutilisables. Mais on met le doigt dans l'engrenage du paradoxe de Russel, si l'on commence à écrire une formule comme héréditaire (héréditaire) : considérer un prédicat R portant sur des prédicats définis par $R(P)$ si et seulement si $\neg P(P)$.

Une solution simple pour éviter cela est de stratifier les prédicats : on aura les prédicats du premier ordre portant sur des termes du premier ordre, les prédicats du second ordre portant sur des prédicats du premier ordre (comme héréditaire) et ainsi de suite. De même $\circ$ est une fonction d'ordre supérieur. On obtient ainsi la logique d'ordre supérieur.

7.2 Introduction du typage

Derrière cette hiérarchie se profile naturellement un système de types. On note $E : A$ le fait que l'expression E est de type A ; on dit parfois dans ce cas que E habite A. $A \to B$ désigne le type des fonctions de A vers B, $A \times B$ désigne le type des couples $\langle a, b \rangle$ avec $a : A$ et $b : B$. On a par exemple true : bool, 2 : nat, not : bool $\to$ bool et plus : nat $\times$ nat $\to$ nat.

Contrôler qu'une expression à un sens consiste du point de vue du typage à vérifier que certaines règles sont respectées : dans une sous-expression de la forme $f(a)$, le type de f est nécessairement de la forme $A \to B$ et a habite A ; $f(a)$ est alors de type B. On a des règles similaires concernant les couples et toutes les structures que l'on peut trouver dans les langages : tableaux, listes, etc.

Si l'on note Prop le type de propositions (*), le type des prédicats sur des entiers sera nat $\to$ Prop, tandis que le type des prédicats portant sur de tels prédicats sera (nat $\to$ Prop) $\to$ Prop. Il n'est donc plus possible d'exprimer le paradoxe de Russel.

(*) Pour des raisons techniques, il peut être souhaitable de distinguer le type des propositions Prop du type booléen bool qui est un type énuméré à deux valeurs.

7.2.1 Curryfication

Considérons maintenant la fonction div rendant le quotient de deux nombres en représentation flottante. On peut donner à div le type float $\times$ float $\to$ float. Mais il peut être intéressant de fixer le premier argument. Par exemple si on le fixe à 1, on obtient la fonction inverse. Il est alors plus élégant de donner à div le type float $\to$ float $\to$ float où la flèche associe à droite, ce qui permet de poser inv = div(1), de type float $\to$ float. Sous cet aspect, div prend en argument un nombre et rend une fonction de float vers float : div est également une fonction d'ordre supérieur. Une telle fonction peut se compiler assez efficacement dans certains langages de programmation dits langages fonctionnels, comme ML.

Mais, pour l'instant, bornons-nous à retenir que $A_1 \to A_2 \to \ldots A_n \to B$ représente le type d'une fonction ayant n arguments de types respectivement $A_1 \ldots A_n$ et rendant un résultat de type B.

Le procédé consistant à remplacer $A \times B \to C$ par $A \to B \to C$ s'appelle la *curryfication*, du nom du logicien Curry qui en a systématisé l'emploi (son invention elle-même remonte à Schönfinkel).

7.2.2 Types dépendants

Souvent on considère que cela n'a pas de sens d'appliquer div à x et y où y est nul. Peut-on intégrer une telle contrainte dans un système de types ?

Pour y répondre, jouons avec les symboles ; ce petit jeu sera en fait légitimé plus loin. Dans les langages de programmation on place d'ordinaire les paramètres entre parenthèses, et on les *nomme*. Procédons de même ici. Au lieu de :

not : bool $\to$ bool et div : float $\to$ float $\to$ float

on notera :

$$\mathrm{not} : (b : \mathrm{bool})\ \mathrm{bool} \quad \text{et} \quad \mathrm{div} : (x, y : \mathrm{float})\ \mathrm{float}$$

le dernier étant en fait une abréviation de :

$$\mathrm{div} : (x : \mathrm{float})(y : \mathrm{float})\ \mathrm{float}$$

On a alors la possibilité d'introduire une condition portant sur y :

$$\mathrm{div} : (x : \mathrm{float})(y : \mathrm{float})(c : y \neq 0)\ \mathrm{float}$$

Autrement dit, $y \neq 0$ est vu comme un type et div admet un troisième argument de ce type. Cela est justifié par la **correspondance** de **Curry-Howard**, qu'il suffira ici d'admettre :

Toute proposition peut être considérée comme un type, dont les habitants sont les preuves de cette proposition.

Explication sommaire : une preuve de $A \Rightarrow B$ permet, grâce à *modus ponens*, de construire une preuve *de B* à partir d'une preuve de *A*, autrement dit c'est une fonction de *A* vue comme type vers *B*. $A \Rightarrow B$ est donc identifié à $A \to B$. De manière analogue une preuve de $(\forall a : A)\, B_a$ permet de construire une preuve de B_a pour tout a donné de A – en général B_a dépend de a. $(\forall a : A)\, B_a$ est donc identifié à $(a : A)\, B_a$, le type des fonctions de A vers B_a où le type du résultat B_a dépend de la valeur de l'argument a. La correspondance de Curry-Howard permet aussi d'interpréter les autres connecteurs logiques.

Lorsque le type d'une composante du type dépend d'une autre composante, on dit que l'on a affaire à un *type dépendant*. Parmi les autres types dépendants dignes d'intérêt, il y a celui des couples $\langle a, b \rangle$ où le type de b dépend de la valeur de a. La notation que nous emploierons ici pour ce type sera $\{a : A \mid B(a)\}$. On ne peut généraliser trop simplement la notation $A \times B$ en $A \times B(a)$, car il faut préciser d'où vient a. Par exemple, on peut représenter le type des nombres non nuls par $\{r : \mathrm{float} \mid r \neq 0\}$, dont les habitants sont des couples $\langle r, p \rangle$ où r est de type float et p est une preuve que $r \neq 0$.

Dans la correspondance de Curry-Howard $(\exists a : A)\ P(a)$ s'interprète exactement comme $\{a : A \mid P(a)\}$. Prouver $(\exists a : A)\ P(a)$ consiste à fournir un témoin a *et* une preuve de p de $P(a)$, autrement dit un « couple dépendant » $\langle a, p \rangle$.

Attention ! $\{r : \mathrm{float} \mid r \neq 0\}$ n'est pas un sous-ensemble de float, contrairement à ce que cette notation représente en théorie des ensembles : c'est un ensemble de couples. Mais si l'on forme l'ensemble des premiers éléments de ces couples, on retrouve bien le sous-ensemble suggéré.

Remarquons que la notation $A \to B$ peut s'employer au lieu de $(a : A)\, B$ chaque fois que B ne dépend pas de a. Voici alors un type possible pour la division :

$$\mathrm{div} : \mathrm{float} \to (y : \mathrm{float})(y \neq 0) \to \mathrm{float}$$

Puisque le type du résultat peut dépendre des arguments, pourquoi s'en priver ?

$$\mathrm{div} : (x : \mathrm{float})(y : \mathrm{float})(y \neq 0) \to \{z : \mathrm{float} \mid x = y \cdot z\}$$

Note : le lecteur exigeant n'aura pas de peine à imaginer une formulation plus réaliste en termes d'encadrement du résultat dans un intervalle. Le but ici est d'illustrer des principes sur un type de données connu et non de discuter des problèmes d'approximation.

Un type différent mais équivalent serait :

$$\mathrm{div} : (x : \mathrm{float})\{y : \mathrm{float} \mid y \neq 0\} \to \{z : \mathrm{float} \mid x = y \cdot z\}$$

On observe que le type est devenu une véritable *spécification*.

7.2.3 Types polymorphes

Revenons à la composition des fonctions. Si l'on se limite aux fonctions entières, le typage de $\circ$ est :

$$\circ : (\mathrm{nat} \to \mathrm{nat}) \to (\mathrm{nat} \to \mathrm{nat}) \to (\mathrm{nat} \to \mathrm{nat})$$

Bien sûr on souhaite aller plus loin et considérer la composition de fonctions entre trois types quelconques α, β et γ. Le plus simple est de considérer α, β et γ comme des variables portant sur des *types* comme nat, bool, float, ou des types plus complexes comme $\mathrm{nat} \times \mathrm{bool}$. On considère que ces types ont eux-mêmes un type noté Set. Cela permet de traiter α, β et γ comme des paramètres ordinaires, ce qui donne le typage suivant pour $\circ$:

$$\circ : (\alpha, \beta, \gamma : \mathrm{Set})\ (\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$$

Ainsi les types polymorphes sont un cas particulier des types dépendants. Set contient en fait tous les types que nous avons introduits précédemment, ceux de div par exemple. De manière analogue, on a vu que toutes les formules logiques habitent un type « frère » de Set appelé Prop. Ainsi un prédicat sur les entiers a pour type $\mathrm{nat} \to \mathrm{Prop}$, ce qui permet d'énoncer le principe de récurrence :

$$\forall P : \mathrm{nat} \to \mathrm{Prop},\quad P(0) \Rightarrow [\forall k : \mathrm{nat},\ P(k) \Rightarrow P(k+1)] \Rightarrow \forall n : \mathrm{nat},\ P(n)$$

7.3 Modes d'utilisation des logiques typées

La présentation qui précède s'appuie sur une des logiques typées les plus puissantes connues à ce jour : le calcul des constructions inductives [6]. On a vu qu'elle permet de représenter de manière uniforme de nombreux ingrédients de la spécification et de la vérification formelle, notamment grâce aux types dépendants et à l'ordre supérieur.

Cette puissance d'expression peut être mise à contribution de deux manières :

- comme cadre général de raisonnement, adapté à la vérification automatique ;
- de manière plus originale, pour obtenir des programmes fonctionnels corrects par construction grâce à une technique d'extraction de programmes.

7.3.1 Cadre logique général

Le calcul des constructions nous a permis d'illustrer la plupart des concepts importants offerts dans le cadre des logiques typées. Les points forts de cette approche sont les suivants :

- la puissance du calcul permet de représenter des systèmes de n'importe quelle complexité ; la capacité d'abstraction offerte, bien employée, permet une expression directe et naturelle (alors qu'une logique moins puissante oblige à effectuer des codages plus compliqués) ;
- le typage introduit de nombreux contrôles de cohérence à l'intérieur même des formules ;
- la logique repose sur une poignée de règles élégantes dont la consistance est assurée.

Minimaliser le nombre et la complexité des règles de base est extrêmement important pour une raison qui dépasse largement la question esthétique. Il devient possible de construire un assistant logiciel à la preuve dont la fiabilité repose entièrement sur un noyau petit et bien délimité, toute démonstration se mettant ultimement sous un format simple traité par ce noyau. On peut donc accorder une confiance extrêmement grande dans les théorèmes démontrés dans un tel outil.

Bien sûr, en pratique, on ne développe pas nécessairement les preuves sous leur forme primitive, et surtout on se donne les moyens de programmer la recherche de ces preuves. Tous les coups sont permis dans cette recherche et dans l'interface avec l'utilisateur. On augmente considérablement la lisibilité en laissant le typage implicite lorsque c'est possible. Ainsi la présentation rapproche progressivement de l'usage, c'est-à-dire de la preuve semi-formelle, sans que la rigueur soit remise en cause. Par exemple, l'opérateur de

composition de deux fonctions $\circ$ tel que nous l'avons décrit plus haut (§ 7.1) possède cinq arguments, les trois premiers désignant les types utilisés par les deux fonctions à composer. Mais la plupart du temps $\circ$ est appliqué sur deux fonctions connues, dont le type est par conséquent connu, comme $\mathrm{triple} : \mathrm{nat} \to \mathrm{nat}$ et $\mathrm{pair} : \mathrm{nat} \to \mathrm{bool}$. Au lieu de $\circ$ (nat, nat, bool, pair, triple), on se permet d'écrire simplement $\circ$ (pair, triple) ou en notation infixe $\mathrm{pair} \circ \mathrm{triple}$.

Quelques outils : un certain nombre d'outils d'assistance à la preuve reposant dans une plus ou moins grande mesure sur les principes qui viennent d'être décrits ont été réalisés depuis la fin des années 70. Plusieurs sont issus de LCF (*Logic of Computable Functions*).

Parmi ceux-ci l'un des plus avancés est sans doute HOL (*Higher Order Logic*) [16], développé à Cambridge. Cependant son système de types est en fait bien plus rudimentaire que celui du calcul des constructions et il repose sur une logique munie de très nombreuses règles. L'un des intérêts de HOL réside dans la richesse des bibliothèques et dans la puissance des mécanismes de recherche de démonstrations qu'il offre.

Les efforts pour outiller le calcul des constructions sont plus récents, cette logique elle-même ne date que de la fin des années 80. Les outils et bibliothèques sont donc moins riches que celles de HOL. Le plus avancé de ces outils, Coq [10] [13], est développé à l'INRIA et à l'ENS de Lyon.

7.3.2 Extraction de programme

On a vu que la correspondance de Curry-Howard procure un cadre homogène pour les fonctions et les démonstrations d'une part, les types et les formules logiques d'autre part. En particulier, le type d'une fonction devient une véritable spécification.

Il est néanmoins trop brutal de suivre cette correspondance au pied de la lettre. Ainsi on observe que la fonction :

$$\mathrm{div} : (x : \mathrm{float})\{y : \mathrm{float} \mid y \neq 0\} \to \{z : \mathrm{float} \mid x = y \cdot z\}$$

a pour second argument et pour résultat des *couples* ⟨nombre, propriété⟩ alors que d'ordinaire on attend des nombres.

Cela convient pour le raisonnement mais pas pour une implantation.

L'*extraction de programme* est le procédé qui vient précisément à point. Il consiste à effacer dans les termes tout ce qui est de nature purement logique, comme $y \neq 0$. Par exemple $\langle r, p\rangle : \{r : \mathrm{float} \mid P(r)\}$ devient $r : \mathrm{float}$. On extrait ainsi une version épurée de la fonction initiale. Dans notre exemple, on retrouverait ainsi :

$$\mathrm{div} : \mathrm{float} \to \mathrm{float} \to \mathrm{float}$$

Plus généralement, d'une fonction de type $(x : X)\ P(x) \to \{y : Y \mid Q(x, y)\}$ on extrait une fonction f telle que $\forall x : X, P(x) \Rightarrow Q(x, f(x))$; cela est garanti par un théorème général de la *théorie de la réalisabilité* qui est la théorie sous-jacente utilisée [27].

On parvient aujourd'hui à développer des programmes non triviaux de cette manière, mais il s'agit encore essentiellement d'une piste de recherche.

8. Conclusion

Les méthodes formelles sont apparues depuis longtemps en milieu académique. L'idée même de vérifier la correction des programmes date des années 60 et remonte même à l'apparition des ordinateurs. Quelques expériences en milieu industriel ont eu lieu durant les années 70, mais sans lendemain faute notamment d'outils de support. En effet, les manipulations formelles requises sont fastidieuses et sujettes à erreur lorsqu'elles sont manuelles.

Actuellement il y a un tournant avec l'arrivée à maturité de plusieurs méthodes, d'outils munis d'interfaces qui se modernisent, et de logiciels d'assistance à la preuve de plus en plus efficaces. Certes les plus puissants d'entre eux sont encore réservés à un cercle restreint de spécialistes.

Mais d'autres, principalement autour des techniques ensemblistes, sont d'ores et déjà utilisés dans l'industrie. On peut tout particulièrement mentionner le cas de B, en France, dans le domaine des transports ferroviaires [7] [2]. Le bagage mathématique nécessaire se limite essentiellement à ce que les ingénieurs ont appris dans leur jeunesse. Une fois n'est pas coutume, félicitons la réforme des mathématiques modernes !

Les années voire les mois qui viennent devraient voir éclore un certain nombre d'expériences, d'évaluations et d'utilisations sur des applications réelles. L'intérêt de ces méthodes ne sera montré que si on les applique avec pertinence : il est aussi peu approprié d'utiliser Z sur des problèmes de contrôle complexe que Estelle pour la structuration d'une base de données. Il faut également savoir estimer le coût de la formalisation en regard du gain attendu, ce qui n'est pas toujours possible faute de données expérimentales.

Terminons par une remarque sur le terme consacré par l'usage de « méthodes formelles ». Il ne s'agit pas de méthodes au sens ordinaire de prêt-à-penser, mais de langages formels et d'outils de support. Le lien avec un cahier des charges informel reste un domaine largement ouvert.

Développement et validation de logiciels. Méthodes formelles

par **Patrick BELLOT**
Département informatique. École Nationale Supérieure des Télécommunications

Jean-Philippe COTTIN
Département informatique. École Nationale Supérieure des Télécommunications

et **Jean-François MONIN**
France Télécom, Centre National d'Études des Télécommunications, Lannion

Références bibliographiques

[1] ABRIAL (J.R.). – *The B-technology.* In FORTE'92, Fifth International Conference on Formal Description Techniques (1992).

[2] BEHM (P.). – *Application d'une méthode formelle aux logiciels sécuritaires ferroviaires.* In Atelier Logiciel Temps réel, 6e Journées Internationales du Génie Logiciel, nov. 1993.

[3] BELLOT (P.). – *Objectif PROLOG.* Masson ed., Paris (1994).

[4] WIRTH (N.) et HOARE (C.A.R.). – *An axiomatic definition of the programming language pascal.* Acta Informatica, 2(4) : 335-355 (1973).

[5] COLMERAUER (A.). – *Prolog in 10 figures.* Communication of the ACM, 28(12) : 1298-1310 (1985).

[6] COQUANT (T.) et HUET (G.). – *A theory of constructions.* In G. Kahn, editor, Semantics of Data Types. Springer Verlag (1985).

[7] DEHBONEI (B.) et MEJIA (F.). – *Formal development of software in railways safety critical systems.* In Railway Oprations, 2 : 213-219, Computa-tional Mechanics Publications (1994).

[8] DIJKSTRA (E.W.). – *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ (1976).

[9] Edinburgh Portable Compilers Ltd. *B-Tool Version 1.1*–User manual/Tutorial/Reference Manual (1991).

[10] CORNES (C.) et al. – *The coq proof assistant reference manual,* v5.10. Technical report, INRIA Rocquencourt, ENS Lyon (1995).

[11] FLOYD (R.W.). – *Assigning meanings to programs.* Mathematical Aspects of Computer Sciences, pages 52-66 (1967).

[12] BIDOIT (M.) et BERNOT (G.). – *Introduction aux spécifications algébriques.* In Actes de l'École des Jeunes Chercheurs du Gréco de Programmation du CNRS, Université de Bordeaux I (1991).

[13] KAHN (G.), HUET (G.) et PAULIN-MOHRING (C.). – *The coq proof assistant, a tutorial,* v5.10. Technical report, INRIA Rocquencourt, ENS Lyon (1995).

[14] COURONNÉ (P.), BERRY (G.) et GONTHIER (G.). – Programmation synchrone des systèmes réactifs : le langage esterel. TSI, 6(4) : 305-315 (1987).

[15] GOOD (D.I.). – *Toward a man-machine system for proving program correctness.* PhD thesis, University of Wisconsin (1970).

[16] GORDON (M.J.C.). – *Hol : A proof generating system for higher-order logic.* In C. Birtwistle and P.A. Subrahmanyam, editors, VLSI Specification, Verification and Synthesis. Kluwer Academic Publishers (1988).

[17] HOARE (C.A.R.). – *An axiomatic basis for computer programming.* Communications of the ACM, 12(10) : 576-580 (1969).

[18] JONES (C.B.). – *Systematic Software Development using VDM,* Second edition. Prentice Hall International (1990).

[19] KING (J.C.). – A program verifier. PhD thesis, Carnegie Mellon University (1969).

[20] McCARTHY (J.). – *Recursive functions of symbolic expressions and their computation by machine.* Communication of the ACM, 3(4) : 184-195 (1960).

[21] MEYER (B.). – *Eiffel : The Language.* Prentice-Hall (1992).

[22] MILNER (R.). – *A proposal for standard ml.* In ACM Conference on Lisp and Functionnal Programming (1987).

[23] MORGAN (C.). – *Programming from Specifications.* Prentice Hall International (1990).

[24] Oxford Science Park, UK .B-Toolkit Beta-Release Version 1.1 – User manual, Reference Manual} (1993).

[25] HALBWACHS (N.), CASPI (P.), PILAUD (D.) et PLAICE (J.). – *Lustre, a declarative language for real-time programming.* In Proc. 10th ACM Symposium on Principles of Programming Languages (1987).

[26] MORGAN (C.) et GARDINER (P.). – *Data refinement of predicate transformers.* In Theoretical Computer Science 87, pages 143-162 (1991).

[27] PAULIN-MOHRING (C.). – *Extraction de programmes dans le calcul des constructions.* PhD thesis, University of Paris VII (1989).

[28] RUSHBY (J.). – *Formal methods and the certification of critical systems.* Technical Report CSL-93-7, SRI International Technical Report, Menlo Park (1993).

[29] LUCKMAN (D.C.), IGARISHI (S.) et LONDON (R.L.). – *Automatic program verification i : a logical basis and its implementation.* Acta Informatica, 4 : 142-185 (1975).

[30] SHANKAR (N.), OWRE (S.) et RUSHBY (J.M.). – *Pvs : a prototype verification system.* In 11th Conference on Automated Deduction (CADE), LNAI 607, pages 748-752. Springer-Verlag (1992).

[31] SHANKAR (N.), OWRE (S.) et RUSHBY (J.M.). – *The PVS Specification Language (Beta Release).* Computer Science Laboratory, SRI International (1993).

[32] SPIVEY (J.M.). – *The Z Notation – A Reference Manual,* Second edition. Prentice Hall International (1992).

[33] WEIS (P.) et LEROY (X.). – *Le langage Caml.* Inter-Éditions (1993).

POUR EN SAVOIR PLUS

Normes internationales

ISO/IEC 8807	1989	Opens Systems Interconnection. A formal description technique based on the temporal ordering of observationnal behavior.
ISO/IEC 9074	1989	Opens Systems Interconnection. A formal description technique based on Extended State Transition Model.
ISO/IEC 10167	1991	Opens Systems Interconnection. Guidelines for the application of Estelle, Lotos and SDL.