Éléments de classification des méthodes formelles

Jean-François Monin	Joseph Sifakis
CNET DTL/MSV	SPECTRE-VERIMAG
Technopole Anticipa	Centre Equation
2 avenue Pierre Marzin	2 rue Vignate
22307 Lannion	38610 Gières

En préliminaire à toute discussion portant sur les avantages et intérêts des différentes méthodes formelles, il est important de bien comprendre ce qu'elles sont et quelles sont leurs principales caractéristiques. Introduisons quelques notions utiles à cet effet.

Par méthode, on entend généralement la marche à suivre pour atteindre un but (Robert). Dans le cadre du développement de logiciels, on cherche, idéalement, à dériver et réaliser un système satisfaisant les exigences exprimées dans un cahier des charges. Plusieurs types de méthodes sont nécessaires. Nous en donnons quelques exemples dans le tableau suivant.

fourniture	*méthode*	*but*
spécifications	synthèse	programme
spécifications + programme	vérification	$\{\text{oui}, \text{non}\}$
programme	compilation	code exécutable
système + objectifs de test	test	jeux de test

Dans le cas de la compilation, il existe pour la plupart des langages des méthodes algorithmiques. En revanche, la synthèse de programmes à partir de spécifications logiques ou la vérification de programmes est généralement un problème dit indécidable, c'est-à-dire ne possédant pas de solution algorithmique : cela ne veut pas dire que ce problème soit insoluble, comme on le croit parfois, mais qu'il est intrinsèquement difficile et qu'il requiert une intervention humaine. Notons d'ailleurs qu'il existe des classes de programmes intéressantes en pratique, où des méthodes de vérification algorithmiques peuvent s'appliquer.

Nous ne chercherons pas ici à discuter de la nature des méthodes. Voici cependant quelques caractéristiques qui nous semblent importantes :

- l'existence d'un langage décrivant l'univers du discours et servant à exprimer les résultats attendus ;
- la possibilité de vérifier la validité des résultats, c'est-à-dire du fait que la méthode conduit bien à une solution du problème initialement posé ;

– la possibilité d'appliquer la méthode d'une manière plus ou moins systématique.

Une méthode est dite *formelle*, si le langage sur lequel elle s'appuie possède à la fois une syntaxe et une sémantique bien définies – mathématiquement définies – de façon à ce que ne subsiste aucune ambiguïté quant à l'interprétation des énoncés exprimés dans ce langage.

Le langage contient les primitives, les constructions et les concepts destinés à l'utilisateur de la méthode. Les méthodes sont définies par un ensemble d'étapes à appliquer à partir d'une description initiale. Il est clair que le langage choisi est souvent déterminant en ce qui concerne la granularité des étapes, le style des descriptions et la nature des solutions. Certaines méthodes, nous l'avons vu, sont algorithmiques : les étapes à effectuer sont entièrement mécanisables et en nombre fini. Les méthodes non algorithmiques comportent des étapes non mécanisées, voire non mécanisables, requérant une intervention humaine. Par exemple, l'utilisateur va deviner et proposer un invariant de boucle. La machine peut dans certains cas vérifier automatiquement que l'expression proposée est satisfaisante. Dans le cas général, appliquer la méthode consiste à faire agir de concert et interactivement un opérateur humain et un outil logiciel. Dans les cas extrêmes, nous avons des méthodes entièrement automatiques ou entièrement manuelles (ou plutôt intellectuelles). Pour appliquer efficacement une méthode, il est nécessaire de rechercher la division optimale du travail entre l'utilisateur et l'outil. Idéalement, on souhaiterait que le premier se borne aux décisions critiques et se décharge des calculs fastidieux sur le second.

La seconde caractéristique est une qualité essentielle de toute méthode scientifique. Elle implique de pouvoir valider l'application de la méthode par rapport à des critères de correction. Pour des méthodes non formelles, ceci peut être vérifié expérimentalement par des tests : étant donné un échantillon de données initiales on peut tester si la méthode aboutit à des résultats corrects. Cela accroît la confiance en la méthode mais n'exclut pas des résultats erronés. En revanche, il est possible d'établir qu'une méthode formelle fournit des résultats corrects pour toute donnée initiale.

La troisième caractéristique exprime une exigence d'efficacité dans l'application de la méthode, la possibilité de reproduire à grande échelle l'expérience de laboratoire. Les articles de recherche mettent généralement l'accent sur les difficultés techniques surmontées par leurs auteurs, il serait bon de montrer plus souvent ce qui peut être systématisé dans la démarche employée. Ceci n'est pas toujours facile à définir car dépend de facteurs subjectifs. Quelques critères sont le temps d'apprentissage de la méthode et l'existence d'outils. Ces dernier jouent deux rôles complémentaires : ils améliorent l'efficacité et empêchent certaines erreurs.

Cette discussion a fait apparaître les principaux éléments constitutifs des méthodes formelles :

– un ou plusieurs langages munis d'une syntaxe et d'une sémantique formelles ; ces langages servent à exprimer des spécifications formelles, des propriétés ou des programmes ;

– des outils permettant de traiter ces textes formels, notamment des outils de *vérification*.

Les outils de vérification servent à démontrer avec la plus grande rigueur possible que le système spécifié est conforme à ce qu'on en attend, «ce qu'on en attend» étant défini par une liste de propriétés ou une autre spécification. Certaines méthodes formelles fournissent des outils de vérification applicables à une forme intermédiaire très proche du code exécutable.

Il existe une grande variété de langages de spécification formelle et de techniques de vérification associées. Les paragraphes qui suivent visent à apporter quelques points de repère, sans prétention à l'exhaustivité. Pour d'autres points de vue et un panorama plus complet, le lecteur intéressé se reportera aux études diverses citées en fin de chapitre.

Notons enfin que, nous mettons l'accent essentiellement sur la spécification et la vérification formelles, qui ont été étudiées en priorité par les chercheurs en méthodes formelles. Les tendances actuelles montrent que la recherche s'intéresse à l'ensemble des axes du développement de systèmes allant de la synthèse à la génération automatique de séquences de test, la génération de diagnostic, le déboguage et les outils d'animation ou de prototypage rapide.

1 Langages de spécification

On appelle langage de spécification un langage utilisé pour exprimer des propriétés ou des comportements attendus d'un système. On peut se poser différentes questions sur un langage de spécification :

– permet-il de se placer au degré voulu d'abstraction ou de détail ?

– permet-il de découper un gros système en modules, de factoriser les développements ?

– possède-t-il une sémantique convaincante (de taille raisonnable) ?

– quels sont les outils disponibles pour effectuer des vérifications syntaxiques et sémantiques ?

Une question est particulièrement discriminante :

– le langage est-il spécialisé à un domaine particulier, ou au contraire est-il à large spectre ? Par exemple, les langages synchrones (Estérel, Lustre, Signal) sont réservés aux systèmes réactifs [1], pauvres en structures de données mais bien armés en structures de contrôle. À l'autre extrême nous avons des langages s'inspirant directement de notions mathématiques générales, par exemple Z qui provient de la théorie des ensembles.

1. Système réagissant en temps négligeable par rapport à leur environnement.

Cette question détermine la division en deux grandes familles de langages, qui fait actuellement l'objet d'un consensus :

– les langages *généraux* : ils s'appuyent sur l'utilisation d'une logique du premier ordre, de la théorie des ensembles (Z, B, RAISE, TLA), de l'algèbre universelle (langages de spécification algébrique, par exemple Larch) ou d'une logique d'ordre supérieur (PVS, HOL, Coq) ;

– les langages *dédiés aux descriptions de comportements.* Il s'agit de langages exécutables. on trouve dans cette classe par exemple les algèbres de processus (Lotos, CCS, CSP), les langages destinés aux protocoles LDS et Estelle (fondés sur des automates), les réseaux de Pétri, les langages synchrones (Estérel, Lustre, Signal), les statecharts. Leur sémantique est souvent exprimée en termes de systèmes de transitions (schématiquement : un ensemble d'états muni d'une relation de transition entre ceux-ci).

2 Techniques de vérification

On peut distinguer deux approches principales à la vérification :

– celle qui met en œuvre un *système d'inférence* c'est-à-dire un ensemble bien défini et formalisé de règles de raisonnement issues de la logique (*modus ponens*, récurrence, substitution de valeurs égales, etc.) ; les outils de support sont des assistants à la preuve (PVS, HOL, Coq, atelier B, LP,...)

– celle qui se fonde sur la construction d'un modèle généralement fini qui représente le système de transition représentant le comportement de l'application modélisée. La vérification des propriétés s'effectue en parcourant exhaustivement ce modèle. Les outils de support sont basés sur le principe de l'évaluation des sp sur un modèle, connu sous le nom de *model checking* (MEC, Cæsar-Aldebaran, Auto, SPIN, Cospan, SMV,...).

Ces approches ont d'abord été exploitées sous une forme primitive. Dans le premier cas, l'utilisateur était amené à indiquer la moindre étape de raisonnement, ne serait-ce que le remplacement de $x + 0$ par x. Dans le second cas, les modèles étaient établis explicitement, par énumération exhaustive des états visités à tarir de l'état initial. Ceci pouvait produire une explosion des besoins en mémoire même pour des systèmes simples. Cependant des progrès considérables ont été effectués.

Concernant la seconde approche, des techniques de représentation implicite ou symbolique des états, permettent de condenser la représentation du modèle. Notons en particulier la technique des BDD (binary decision diagrams) pour la représentation symbolique des fonctions booléennes, qui trouve des applications intéressantes à la vérification des systèmes à états finis. Malgré leurs limitations intrinsèques, et grâce à des abstractions judicieuses, les techniques par model checking ont été appliquées à des systèmes réalistes dans le domaine des composants matériels ou des protocoles de communication.

Dans les outils logiques, de nombreuses démonstrations, plus ou moins simples, peuvent être construites automatiquement, en logique propositionnelle bien sûr mais aussi dans certains fragments d'algèbre, de logique du premier ordre, d'arithmétique, de théorie des automates, etc. L'utilisateur peut alors se concentrer sur l'architecture générale de sa preuve et laisser les détails fastidieux à la machine.

Notons aussi que les démonstrations construites avec l'assistance de ces outils peuvent toujours être contrôlées par un être humain (ou à l'aide d'un logiciel différent). En revanche, si on vérifie une propriété au moyen d'un outil entièrement automatique, le raisonnement sous-jacent est perdu.

3 Deux grandes familles de techniques

Les langages de spécification généraux, qui sont basés sur une logique, sont assez naturellement associés à une technique de vérification par inférences, tandis que les langages comportementaux sont associés à des outils de vérification par construction de modèle.

Cette distinction, qui s'est constituée historiquement, et que nous conserverons par commodité pour l'analyse, ne doit pas être considérée comme absolue : des recherches actuelles ont pour objectif d'intégrer ces approches. Voyons quelques atouts et difficultés propres à chaque famille.

3.1 Capacité d'expression et de modélisation

La seconde famille est surtout adaptée à la description des problèmes de contrôle et elle ne vise pas un traitement formel des structures des données. La première famille ne présente pas de limitations *a priori* concernant la nature des domaines qu'elle peut décrire mais elle ne fournit pas de constructions spécifiques pour la description des comportements. Lorsque l'on souhaite exprimer des spécifications logiques dans le cadre de la seconde approche, on utilise souvent une logique modale, par exemple, logique temporelle, μ-calcul. Il est nécessaire alors de définir la sémantique des formules de la logique dans le modèle du langage de description des comportements.

3.1.1 Capacité d'abstraction

Il est important, au cours d'une modélisation, de pouvoir oublier les détails non significatifs ou de s'exprimer à un grand degré de généralité. Pour permettre cela un langage de spécification formelle doit offrir des mécanismes d'abstraction. Les langages de la seconde famille permettent de construire des comportements à partir de comportements élémentaires et des structures de contrôle. Les mécanismes d'abstraction se limitent à l'encapsulation. Les langages logiques ont une pouvoir d'abstraction supérieur dans la mesure où ils permettent de décrire les spécifications à partir de propriétés générales et au moyen de notions dont le pouvoir d'abstraction a été éprouvé jusqu'en mathématiques : ensembles, prédicats, relations, fonctions, quantifications logiques, catégories. Cette abstraction a parfois un prix. Il arrive que l'on définisse un objet en le caractérisant par ses propriétés. Il est alors nécessaire de s'assurer de l'existence d'un tel objet, ce qui n'est pas toujours trivial.

3.1.2 Compositionnalité

Il faut distinguer la compositionnalité dans la spécification et dans la vérification.

Les méthodes formelles majeures fournissent toujours des moyens de décomposer une spécification. Toutefois, le type de décomposition n'est pas le même pour les spécifications logiques et comportementales. Les spécifications logiques consistent en un ensemble de formules, les spécifications comportementales en un ensemble de processus ou modules qui coopèrent. On peut modifier ou écrire les spécifications logiques de façon incrémentale, ajouter ou enlever une formule sans qu'une modification du reste des spécifications soit nécessaire. Ceci est un avantage par rapport aux spécifications comportementales.

La vérification est par nature compositionnelle dans la première approche en ce sens que la vérification des propriétés d'un système composé s'obtient à partir des propriétés de ses composants. Pour la seconde famille, la méthode de vérification n'utilise pas la compositionnalité dans la mesure où on aplatit la description (détruit la structure) pour obtenir le modèle à vérifier. Toutefois, on peut combiner model checking et méthodes de vérification compositionnelle.

3.2 Facilité d'utilisation

Les phénomènes de culture jouent un rôle important dans la pénétration des langages de spécification. Il n'est guère étonnant d'observer que ceux qui connaissent le plus de succès s'apparentent à des langages de programmation. Ainsi SDL est utilisé couramment dans le milieu des télécommunications, contrairement à Lotos, et B semble plus accessible que Z alors que leurs bases théoriques sont les mêmes.

On peut raisonnablement estimer que plus les notions d'un langage de spécification sont concrètes – proches de l'expérience des utilisateurs – plus ce langage est facilement accessible. Cela ne signifie pas nécessairement que ce dernier est plus facile à utiliser. En effet, rédiger une spécification formelle comprend deux types d'activité : une activité de modélisation et une activité de formalisation. Seule la seconde est facilitée lorsque le langage est plus accessible. Les deux tâches sont liées dans la mesure où la modélisation fait intervenir les notions du formalisme utilisé, mais ce qui revêt le plus d'importance est la nature même de l'application visée, qui décide une grande partie des concepts à décrire : il faut, par exemple, identifier les objets soumis à traitements, les événements, les réactions qu'ils provoquent, les composants physiques. Dans tous les cas, il est des aspects que le modélisateur choisit de *ne pas* représenter : inutile de descendre au niveau du transistor lorsqu'on parle d'une bascule D ; dans le problème de l'échiquier mutilé [2], il est superflu de considérer les coordonnées des cases. Ces oublis volontaires – ou *abstractions* – sont essentiels dans le processus de modélisation. Les deux exemples précédents illustrent que certaines étapes d'abstraction sont triviales alors que d'autres le sont moins. Les principales dépendent uniquement du problème à résoudre. S'y ajoutent celles qui proviennent de l'impossibilité, théorique ou pratique, de représenter certains aspects dans le formalisme choisi. Par exemple, certains langages d'automates permettent difficilement la représentation du contenu des messages convoyés par un protocole.

En résumé, il faut garder une certaine prudence avant d'affirmer que tel formalisme est plus facile d'utilisation que tel autre. Ce que l'on peut dire est qu'un langage conçu pour une classe d'applications donnée a d'excellentes chances de permettre d'emblée une expression au bon niveau d'abstraction. C'est précisément le cas dans la seconde famille de techniques, qui fournit de bons outils conceptuels pour traiter les systèmes composés d'entités qui interagissent, et dans lesquels la complexité se situe dans le séquencement des communications.

Cette discussion pourrait se transposer du niveau de la spécification à celui de la vérification, mais il s'avère qu'une part non négligeable de la difficulté consiste à déterminer les propriétés à vérifier. Cette étape est toujours assez technique et nécessite une bonne capacité d'abstraction, quelle que soit l'approche. Elle se concrétise sous des formes variées : invariants à respecter, automates représentant un aspect du comportement, formules de logique temporelle, etc.

En ce qui concerne la vérification, les outils de la seconde famille sont beaucoup plus faciles à utiliser, puisque l'interaction avec l'utilisateur est minimale. En revanche, démontrer une propriété non triviale dans une approche de la première famille requiert parfois beaucoup de travail. Ce dernier est de plus en plus automatisé grâce à l'introduction, plus ou moins avancée selon les outils, de procédures de décision. Néanmoins on ne peut espérer atteindre le degré d'automatisation des outils de la seconde famille. *Il faut en effet choisir entre capacité de description et décidabilité.*

L'analyse des échecs se présente également de manière différente. Dans l'approche logique, l'impossibilité de démontrer une conjecture doit être analysée pour remonter à la source du problème, alors que les outils de la seconde famille fournissent des contre-exemples sous forme de scenarii relativement faciles à exploiter.

3.3 Facteurs limitatifs

Toutes les méthodes formelles se heurtent à des difficultés dès que l'on veut aborder des applications de grande taille. Dans les approches logiques, il s'agit de produire efficacement un très grand nombre d'étapes de raisonnement. Dans les approches qui consistent à parcourir exhaustivement un modèle fini, l'obstacle est la croissance exponentielle de la taille de ce dernier.

2. Peut-on paver, par des dominos, un échiquier dont on a retiré les case A1 et H8 ?

Dans toutes les approches on fait reculer les limites en employant des abstractions pertinentes. Au-delà, chacune dispose d'armes spécifiques : utilisation de la compositionnalité et introduction de procédures de décision pour les approches logiques, codages efficaces des modèles (quotients par une relation d'équivalence, partage de données) pour la seconde famille.

4 Précautions dans la mise en oeuvre des outils

Dans la mesure où les méthodes formelles sont utilisées sur des systèmes critiques, il est important d'avoir l'assurance que les outils employés dans la chaîne de production, outils de vérification et de génération de code notamment, rendent bien le service attendu. Bien entendu tous sont intensivement testés sur des cas très variés, mais il est nécessaire de compléter la confiance expérimentale ainsi obtenue, par l'adoption d'un soin particulier dans leur développement. À cet effet ce dernier doit être basé sur une approche formelle.

Il faut exprimer la théorie mise en œuvre au moyen de concepts mathématiques généraux, et représenter les structures de données et les algorithmes complexes utilisés dans les outils. Pour ce faire, il est nécessaire de faire appel à des techniques de preuve.

Pour rendre praticable cette approche, des outils ont été conçus autour d'un noyau minimal de langage et utilisent des formats intermédiaires qui ont une sémantique bien définie. Ce principe a été suivi pour réaliser aussi bien des compilateurs que des outils de preuve et des model checkers.

Par exemple, les compilateurs des langages Esterel et Lustre utilisent des formes intermédiaires qui déterminent un langage noyau vers lequel une méthode de traduction formelle du langage source a été étudiée. Le compilateur Cæsar génère à partir d'une description Lotos un réseau de Pétri étendu avec des données. Les systèmes Coq et HOL, en suivant les principes de structuration de LCF, sont architecturés autour d'un noyau algorithmique relativement petit dont l'unique objectif est de contrôler qu'un objet, prétendu représenter une démonstration, respecte les règles d'inférence d'un noyau logique restreint.

Cette approche de construction permet de garantir la qualité des outils, dans la mesure où ils implémentent des transformations prouvées correctes. L'utilisation de formes intermédiaires permet des extensions et des interconnexions sans nuire à l'intégrité des outils.

Références

[1] D. Craigen, S. L. Gerhart, and T. J. Ralston. An international survey of industrial applications of formal methods. Technical Report NIST GCR 93/626-V1 & 2, Atomic Energy Control Board of Canada, US National Institute of Standards and Technology, and US Naval Research Laboratories, 1993.

[2] Formal Methods Working Group. Formal methods: State of the art and future directions. Proceedings of the ACM Workshop on Strategic Directions in Computing Research, June 1996.
http://www.cs.cmu.edu/afs/cs/usr/wing/www/mit/mit.html.

[3] Gérard Huet. Certification du logiciel : méthodes et outils. Rapport de fin de contrat 11, Premier ministre, SGDN, 1994.

[4] Jean-François Monin. *Comprendre les méthodes formelles, panorama et outils logiques.* CTST. Masson, 1996. ISBN 2-225-85304-5.

[5] John Rushby. Formal Methods: Instruments of Justification or Tools for Discovery? In *Nordic Seminar on Dependable Computing System 1994*, Department of Computer Science, August 1994. The Technical University of Denmark.

[6] John Rushby. Formal methods and their role in the certification of critical systems. Technical Report CSL-95-01, Computer Science Laboratory, Menlo Park CA 94025 USA, March 1995.

[7] Joseph Sifakis. Formal methods and their evaluation. In *Formal Design of Safety Critical Embedded Systems*, April 1997.