. Editorial

. Spécification formelle et synthèse de programmes

Le faible niveau d'abstraction des langages de programmation actuels est en grande partie responsable des problèmes que connaît le logiciel.
Pour remédier à cet état de fait, la programmation automatique propose d'une part de ramener l'expression de la spécification d'un programme à un niveau purement mathématique, et d'autre part de construire des générateurs permettant de synthétiser automatiquement, à partir de ces spécifications, des programmes écrits dans des langages classiques. Dit autrement, on se contente dans la spécification d'exprimer le " Quoi ", le générateur de programmes produisant le " Comment ", c'est à dire la stratégie de calcul, l'algorithme, ainsi que le programme final avec tous ses détails.
Les recherches dans ce domaine, si elles ne sont pas achevées, ont atteint une maturité suffisante pour autoriser un transfert vers l'utilisation industrielle. Menées depuis plus de 20 ans, elles tirent leur inspiration et leurs méthodes de deux grands courants de l'informatique, la logique – sous l'appellation plus connue ici de " méthodes formelles " – et l'Intelligence Artificielle. Destinées au départ à la vérification de programmes, notamment par une preuve formelle, ce n'est que depuis quelques années que ces recherches se sont orientées vers la synthèse automatique.
Ce numéro d'Imagine décrit les principales approches et méthodes utilisées en synthèse de programmes et fait le point sur les expériences menées dans ce domaine à la DER.

Editorial

Même si de gros progrès ont été accomplis depuis le temps où dialoguer avec la machine consistait à aligner des suites interminables de "0" et de "1", les langages de programmation actuels restent très proches de la machine et imposent une description totale, à un très bas niveau d'abstraction, du processus permettant d'obtenir le résultat escompté.
Les conséquences de cet état de fait sont un coût de conception élevé, une validation des programmes difficile, particulièrement lorsqu'un haut niveau de sûreté est exigé, et une évolution et une maintenance des applications particulièrement ardues et contraintes.
La programmation automatique a pour but d'offrir des langages formalisés, en fait reposant sur les notions mathématiques élémentaires, et indépendants de toute considération relative à la machine. D'une part, cela permet au spécifieur d'avoir une expression claire et concise de ses besoins. D'autre part, la sémantique des mathématiques étant claire et bien connue, contrairement à celle des langages de programmation, il devient possible de mettre en œuvre des générateurs de programmes à partir des spécifications. Ces générateurs sont des sortes de super-compilateurs intégrant de nombreuses connaissances expertes dans le domaine de la programmation. Cependant, leur réalisation est bien plus complexe que celle des compilateurs ordinaires, et commence seulement à être suffisamment maîtrisée pour permettre des implantations utilisables.
Le système Descartes en fin de développement à la DER est ainsi l'un des rares générateurs de programmes non dédié à un domaine d'application spécifique, et le premier qui soit entièrement automatique.

J-L.Dormoy & B. Ginoux

SPECIFICATION FORMELLE ET SYNTHESE DE PROGRAMMES

1. LA CRISE DU LOGICIEL

1.1 Les Faits

Le logiciel est actuellement confronté à des problèmes qui concernent l'ensemble des phases du fameux cycle de vie.
On peut schématiquement regrouper ces problèmes en trois grandes catégories qui sont le développement, la validation et la maintenance.

. Les problèmes de développement
La programmation est une activité qui nécessite de multiples considérations très techniques et ce, même lorsque les programmes à écrire sont conceptuellement simples.

Toutes ces considérations techniques sont bien entendu autant de sources d'erreurs potentielles et rendent difficiles l'écriture des programmes ainsi que leur mise au point. La programmation est donc une activité longue et de ce fait, coûteuse, même à un niveau élémentaire. Le problème se complique évidemment du fait de la taille des applications.

. Les problèmes de validation
Une fois écrits, les programmes doivent être validés. En effet, le programmeur n'étant qu'un être humain il a évidemment pu se tromper et ce d'autant plus qu'il a traité deux problèmes à la fois: la résolution du problème demandé d'une part, c'est à dire l'aspect algorithmique et d'autre part la mise en oeuvre du mécanisme de résolution dans un programme, c'est à dire l'aspect purement programmation. Les erreurs peuvent se situer aux deux niveaux.
De plus, un programme n'est pas écrit isolément, et en général le contexte de l'application où le morceau de programme doit s'insérer est mal connu ou mal précisé. Bref, qui a déjà écrit un programme sait qu'il n'est jamais juste du premier coup. Or, la validation de programme reste difficile et mal connue.

Il existe un continuum de méthodes situées entre deux approches majeures : l'approche " jeux de tests ", qui, si elle garantit normalement les cas testés, reste empirique et insatisfaisante en général, et l'approche idéale " validation formelle " qui tente de prouver formellement que le programme vérifie un certain nombre de propriétés jugées importantes. Si l'approche " validation formelle " est bien plus satisfaisante théoriquement, elle est peu utilisée dans la pratique car il existe peu de techniques de validation formelle opérant sur les langages de programmation les plus couramment utilisés. D'autre part, même lorsque l'on a utilisé des langages plus adaptés (plus abstraits), les capacités des démonstrateurs de théorèmes sur lesquels cette approche repose restent insuffisantes dans le cas général.
Une technique intermédiaire est par exemple *d'instrumenter* le programme par des instructions vérifiant à l'exécution qu'une condition est remplie, agrémentée éventuellement d'actions d'exception. Ces instructions sont, si programme et données sont correctes, *mortes*, c'est-à-dire ne sont jamais exécutées.
En pratique, on se contente souvent de l'approche " jeux de tests " et cela pose un réel problème pour les logiciels critiques. Mais quelle que soit l'approche, une aide automatisée est rendue difficile par le faible niveau des langages.

. Les problèmes de maintenance
De même que précédemment, qui a déjà maintenu un programme sait qu'il est difficile de conduire en parallèle l'évolution de la documentation et du programme correspondant, et que cette documentation est souvent incomplète ou incorrecte. Il faut souvent aller au coeur du code source, en découdre directement avec les " astuces " du programmeur précédent et si possible tenter de supprimer plus de bugs que l'on n'en introduit.
Faire évoluer un programme est également un problème ardu. Même dans le cas où cela est possible, c'est souvent tellement difficile que cela n'en devient souhaitable ni sur le plan technique, ni sur le plan financier.
Ceci est dû à l'extrême rigidité des langages de programmation actuels: une modification conceptuellement minime peut entraîner de lourdes répercussions sur l'ensemble du programme. Il suffit par exemple qu'elle touche à une structure de données très utilisée dans le programme.
On pourrait envisager une modification automatique du source de l'application, mais cela nécessite une compréhension approfondie

du programme, qui n'est pas possible dans l'état actuel des connaissances avec le faible niveau d'abstraction des langages.

1.2 Les Causes

De nombreuses raisons peuvent sans doute être avancées pour expliquer la crise actuelle du logiciel.
L'une de celles-ci, majeure, réside dans le **faible niveau d'abstraction des langages de programmation actuels, qui interdit aussi bien une expression claire des besoins qu'une aide automatisée à la conception, la validation et la maintenance**.
Ces langages reposent sur des concepts qui sont beaucoup trop près de la machine et beaucoup trop loin du programmeur. Ils obligent à décrire les programmes en termes d'opérations élémentaires de très bas niveau, par exemple, en termes de stockage et d'écrasement de valeurs dans des " cases mémoire ".
De plus, s'ils sont opérationnellement proches de la machine, leur forme rend tout traitement automatisé autre que leur compilation très difficile.

2. LA SYNTHESE DE PROGRAMMES

2.1 L'idée

La Synthèse de Programmes est l'une des réponses possibles au problème précédent. En effet, élever le niveau d'abstraction des langages de programmation suppose la capacité de retraduire automatiquement les programmes écrits dans ces nouveaux langages en programmes écrits dans des langages que l'ordinateur comprend. En d'autres termes, cela signifie automatiser, le plus possible, l'activité de programmation.
Le terme " Synthèse de Programmes " désigne de façon générale **l'ensemble des tentatives concernant l'automatisation de tout ou partie du processus de programmation.**

2.2 Perspective historique

Cette définition, pour satisfaisante qu'elle puisse paraître au premier abord laisse pourtant bien des libertés au concept. En effet, celui-ci n'est défini que par rapport à la notion de "processus de programmation", notion en perpétuelle évolution.

La première étape majeure de cette évolution, remonte au temps où les premiers langages assembleurs délivrèrent les programmeurs de la nécessité de s'entretenir avec l'ordinateur directement en hexadécimal, voire en binaire. La deuxième étape majeure fut l'arrivée en 1957 du premier compilateur FORTRAN. Les programmeurs purent alors écrire leurs programmes dans des langages "de haut niveau", laissant au compilateur le soin de transformer leurs instructions élaborées opérant sur des structures de données complexes en instructions de base manipulant des registres et des adresses mémoires.
De même, plus récemment, les langages fonctionnels, logiques, orientés objets ou " à base de règles " ont encore élevé, chacun à leur manière, le niveau d'abstraction des programmes permettant au programmeur de s'affranchir d'une partie de son travail de programmation.

La notion de processus de programmation est donc changeante et n'a de sens véritable que par rapport à une époque donnée. De la même façon, le terme "Synthèse de Programmes" désigne simplement l'ensemble des tentatives visant à automatiser tout ou partie du processus de programmation **qui a cours à une époque donnée**. On peut donc légitimement considérer que les langages assembleurs puis les premiers compilateurs furent à leurs époques respectives les premiers systèmes de programmation automatique !

L'histoire de la programmation automatique et l'histoire de la programmation ne sont donc qu'une seule et même histoire, **la programmation " automatique " d'aujourd'hui n'étant finalement que la programmation "normale" de demain**. La programmation automatique apparaît comme

une cible mouvante qui recule de quelques pas à peine croît-on l'avoir atteinte.

2.3 L'apport des méthodes formelles et de l'Intelligence Artificielle

Il y a une vingtaine d'années, Hoare et Dijkstra lançaient un cri d'alarme et un appel. Le cri d'alarme concernait la faible fiabilité des programmes, notamment lorsqu'ils peuvent avoir des effets sur la sûreté. L'appel visait à initier un effort de recherche sur la fiabilité des programmes.
La direction de recherche essentielle était la *preuve* de programmes. Il s'agissait de fournir, outre le programme, une spécification de propriétés que l'on pensait vérifiées par le programme. Cette spécification était naturellement exprimée dans le langage mathématique, puisqu'on visait des preuves. Il fallait en outre définir un lien entre la spécification mathématique, dont la sémantique est claire et connue, et le programme, théoriquement beaucoup plus obscur. L'idée principale consistait à définir des *invariants*, c'est-à-dire des propriétés conservées par des quantités impliquées dans des boucles. Le problème était de prouver que ce sont bien des invariants.
Notons que l'objectif était de prouver les parties jugées critiques d'un programme, et il semblait inaccessible de le spécifier complètement.
Diverses notations et langages systématiques sont apparus alors, notamment Z et VDM, et les spécifications algébriques. Tous reposent sur le langage mathématique ou logique, et expriment la fonctionalité essentielle du programme, mais sans référence à la machine. Les exemples de spécification se multipliant, la démonstration était faite que l'on pouvait raisonnablement spécifier mathématiquement une large classe de programmes.
Par contre, le traitement automatique — ici la preuve — était et est resté loin derrière. La démonstration de théorèmes, un des sujets de prédilection de l'IA, reste du domaine de la recherche. Encore aujourd'hui, un démonstrateur de théorèmes est un outil *vérifiant* qu'une preuve est correcte, la *découverte* des étapes essentielles de la preuve étant laissée à l'utilisateur humain. Prouver un programme complet est donc une activité longue et pénible; pour autant, la preuve de programmes est aujourd'hui utilisée dans l'industrie pour certains logiciels critiques.

Cette recherche a cependant eu une conséquence inattendue; il s'est avéré que l'on pouvait spécifier mathématiquement l'intégralité d'un programme. Or, le niveau mathématique constitue une rupture qualitative avec les langages de programmation proposés depuis le début de l'informatique. Pour la première fois, un programme est " exprimé " sans référence aux constituants de la machine (séquentialisation des instructions — même sur une machine parallèle — et emplacements mémoire). Et il a fallu en quelque sorte le détour par la preuve de programmes pour oser franchir ce pas. D'ailleurs, d'autres tentatives pour augmenter le niveau des langages à partir des langages existants — par exemple le langage SETL, sorte d'Algol incluant un type de données ensembliste — n'ont pas su franchir ce pas.
Pourquoi dès lors être contraint d'ajouter à la spécification du programme le programme lui-même ? De plus, même si en théorie la génération et la preuve de programmes sont deux problèmes équivalents, le développement des systèmes à bases de connaissances en Intelligence Artificielle a rendu plus plausible la réalisation d'un générateur de programmes. De façon surprenante a priori, il est plus simple à un système automatique de synthétiser entièrement un programme plutôt que de prouver qu'un programme donné réalise une spécification donnée! En effet, prouver qu'un programme donné réalise une spécification donnée consiste à établir que ce programme est l'une des innombrables façons de programmer cette spécification. Alors que générer le programme implique d'en synthétiser une, à la forme fréquemment très standardisée.
Pour autant, comme on va le voir, nous ne sommes qu'à l'aube de la programmation automatique à partir de spécifications mathématiques.

3. ETAT DE L'ART

3.1 Les approches

On peut classer les approches par rapport à deux grands critères:

- **la généralité du système**: le langage de spécification et le générateur de programmes sont-ils dédiés ou non à un domaine d'application spécifique?

- **le degré d'automatisation du système**: le générateur est-il entièrement automatique ou coopère-t-il de manière interactive avec l'utilisateur?

. les systèmes dédiés

Les systèmes dédiés sont bien entendu les plus fréquents car beaucoup plus faciles à construire. Le domaine d'application étant spécifique, le langage de spécification peut offrir de manière prédéfinie les primitives pertinentes du domaine. La génération de code peut, elle aussi, être optimisée en fonction du domaine considéré et souvent (mais pas toujours), le système est complètement automatique.

On trouve des systèmes de programmation automatique dédiés, en particulier dans les domaines de l'interrogation de bases de données (SQL), des interfaces graphiques (générateurs d'écrans,....), de la résolution de problèmes combinatoires (SIREN, DESIGNER-SOAR), ...

. les systèmes généraux

Les systèmes généraux sont beaucoup plus rares, car bien sûr plus complexes. Ils s'adressent potentiellement à tous les domaines d'applications et ne peuvent par conséquent intégrer les primitives pertinentes de tel ou tel domaine particulier.

Dans la mesure où ces systèmes sont plus complexes et où le type de programme à générer n'est pas connu à l'avance, nombre d'entre eux sont *semi-automatiques*. C'est à dire qu'ils sont utilisés de manière interactive par l'utilisateur qui s'en sert en fait comme de boîtes à outils ou, selon le vocabulaire consacré, comme *d'assistants de programmation*.

L'essentiel des travaux de recherche dans le domaine des systèmes généraux, mis à part les premières tentatives de Manna et Waldinger vers la fin des années 70 puis le système PECOS/LIBRA de Barstow et Kant au début des années 80, concerne les systèmes semi-automatiques. Le plus célèbre et le plus achevé est le système KIDS, réalisé par le Kestrel Institute, un institut privé situé dans les locaux de Stanford et financé par l'armée américaine.

Par ailleurs, un certain nombre de langages de spécification formels généraux, tels que les langages de spécification algébriques comme PLUSS, CLEAR, OBJ-3,.... ou des langages de type mathématique comme le langage Z ou plus récemment le langage B d'Abrial sont plutôt utilisés pour prouver des programmes et n'ont pas donné lieu pour l'instant à la réalisation de générateurs de programmes.

3.2 Les méthodes

Il existe quatre grandes catégories de méthodes pour construire des systèmes de programmation automatique, à savoir: les méthodes procédurales, déductives, inspectionnelles, et transformationnelles.

. Les méthodes procédurales

Ce sont les méthodes conventionnelles utilisées dans les compilateurs. On écrit **des programmes classiques dans des langages procéduraux**. Le problème fondamental de ce type de méthode réside dans son **incapacité à utiliser correctement de grandes quantités de connaissances**.

Or, dès que l'on vise un système "général" et non un système dédié à un domaine d'application très spécifique et très restreint, on se heurte très rapidement à ce problème d'utilisation de la connaissance, notamment dans les domaines du raisonnement, de la logique et des manipulations symboliques.

Les méthodes procédurales, mélangeant inextricablement connaissances et structures de contrôle algorithmiques, engendrent des systèmes très difficiles à mettre au point, à valider et à maintenir (rajouter une fonctionnalité relève parfois du tour de force).

En fait elles possèdent elles-mêmes tous les inconvénients des langages procéduraux puisque

pour construire des langages qui les remplacent, elles les utilisent.

. les méthodes déductives

Elles consistent à construire un programme à partir de **dérivations de preuves établies par un démonstrateur de théorèmes**.

L'idée repose sur le fait que le problème de la synthèse d'un programme satisfaisant une spécification donnée est formellement équivalent au problème de la recherche d'une preuve constructive de la satisfiabilité de la spécification. La spécification se présente donc sous la forme d'une formule logique à démontrer, le mécanisme de déduction pouvant être a priori quelconque (par exemple la résolution).

Ce type de méthode souffre des limitations induites par les capacités (même actuelles) des démonstrateurs de théorèmes. Il y a en fait deux problèmes principaux: l'incapacité à contrôler efficacement le processus de recherche de la preuve qui est, par nature, infini et le fait que rien ne permet d'affirmer, dans le cas où le démonstrateur engendre effectivement une preuve, que celle-ci correspond au chemin le plus court entre les faits initiaux et le théorème à démontrer.

La méthode déductive est donc pour l'instant condamnée à résoudre des cas relativement simples. Pourtant elle reste intéressante dans la mesure où elle est extrêmement générale. La méthode déductive pourrait jouer un rôle important dans les futurs systèmes de programmation automatique.

Le système expérimental Coq de l'INRIA met en oeuvre cette méthode.

. les méthodes inspectionnelles

Elles sont de plus en plus utilisées mais le plus souvent de manière complémentaire, c'est-à-dire, en plus d'une autre méthode.

Ces méthodes consistent à identifier des schémas formels de spécification ou "clichés" et à déterminer l'implémentation correspondante en choisissant dans un catalogue de schémas formels d'implémentation. L'avantage de ces méthodes est qu'elles sont relativement proches de la façon dont raisonnent généralement les programmeurs. L'inconvénient est qu'elles ne peuvent fonctionner que sur des cas relativement simples, sachant que sur ce type de cas, elles sont particulièrement efficaces. C'est pourquoi la tendance actuelle est de combiner ce type de méthodes avec les méthodes déductives ou transformationnelles de manière à pouvoir également prendre en compte les cas plus compliqués en utilisant des mécanismes généraux de transformation ou de déduction.

Le principal système utilisant ce type de méthodes est "The Programmer Apprentice" de Rich et Waters au MIT.

. les méthodes transformationnelles

Ce sont les méthodes les plus utilisées actuellement. Elles consistent à effectuer sur une spécification écrite généralement dans un langage de très haut niveau, une **séquence de transformations graduelles** dont l'effet est de se rapprocher puis d'atteindre une spécification de bas niveau, c'est-à-dire un programme. Chaque transformation est élémentaire et modifie l'état courant de la spécification de façon très progressive, à la différence des clichés utilisés dans les méthodes inspectionnelles qui permettent eux d'installer d'un seul coup des paradigmes de programmation potentiellement très complexes. Ce sont les multiples combinaisons possibles de ces transformations élémentaires qui font que les méthodes transformationnelles sont plus générales que les méthodes inspectionnelles.

Chaque transformation correspond en fait à un **raffinement progressif d'un programme abstrait**. Un raffinement peut par exemple être obtenu par différentiation formelle, tabulation, généralisation ou spécialisation, dérécursivation, fusion de boucles, tupling, génération d'une structure de contrôle explicite, passage de structures de données abstraites à des structures de données existant dans le langage cible,... Certaines de ces transformations peuvent aussi correspondre à des opérations d'optimisation, à un état donné de raffinement du programme (ce sont alors des transformations horizontales, par opposition aux transformations verticales qui font descendre d'un cran dans l'abstraction).

Chaque transformation **préserve la correction** du programme dans la mesure où le "morceau" de programme remplacé et celui qui le remplace sont logiquement équivalents. La correction du programme final est donc assurée à condition

bien sûr que la spécification initiale soit elle-même correcte.

. à l'avenir : la théorie des catégories ?

La théorie des catégories est une théorie mathématique, appelée aussi parfois " algèbre universelle ", qui est en quelque sorte la " théorie des théories " et des liens qu'elles entretiennent. Elle est par exemple omni-présente en topologie algébrique. Depuis quelques années, certains théoriciens se sont aperçus que l'on pouvait exprimer les problèmes généraux de l'informatique en termes de catégories. On cherche aujourd'hui comment les catégories pourraient servir de langage de spécification — par exemple via la théorie des esquisses. L'attrait immédiat des catégories est qu'elles reposent sur les notions d'objets et de flèches, et donc descriptibles graphiquement. En revanche, les concepts de base de cette discipline réclament un certain effort d'assimilation…

Le premier système opérationnel utilisant les catégories, SPECWARE du Kestrel Institute, doit sortir prochainement.

4. LA PROGRAMMATION AUTOMATIQUE A LA DER

4.1 CLIM,KSE, EUGENIE,SIREN, ...

L'intérêt de la programmation automatique a été découvert empiriquement au fil des applications et des années. L'expérience rapportée ici est celle du Groupe Intelligence Artificielle au cours des années 85-95.

La première expérience de programmation automatique concernait un système expert de vérification de cohérence des données d'entrée de CLIM 2000, système de calcul de thermique du bâtiment, développé à AEE. Pour des raisons de difficulté de portage du moteur d'inférence sur certaines machines, un compilateur de contraintes de cohérence dédié a été réalisé, qui remplaçait le système expert par un programme C fonctionnellement équivalent.

La deuxième grande application a été KSE, un système expert de traitement des alarmes électrique en ligne d'une centrale REP, construit à IMA-TIEM puis à EP-CCC. Ce système était basé sur trois grandes catégories de connaissances : la topologie de l'installation, les modèles de fonctionnement des composants élémentaires, et une stratégie de diagnostic. Chaque catégorie de connaissances était décrite séparément, et une série de générateurs de code (ici, des règles élémentaires étaient générées) opérationnalisaient ces connaissances en un système temps réel. L'acquisition des connaissances, c'est-à-dire la conception du système, leur validation et leur évolution étaient ainsi rendues possibles. Le système temps réel produit comportait environ 50 000 règles, toutes générées.

Une troisième application a été EUGENIE, un système expert de génération de procédures CLIST pour les IBM du Centre de Calcul. L'avantage des programmes générés était qu'ils contenaient systématiquement toute une série de vérifications.

Le système SIREN générait un programme à partir de l'énoncé d'un problème combinatoire quelconque. Il a été utilisé notamment dans le calcul des plans de rechargement des centrales REP.

Mis à part SIREN, ces générateurs de programmes étaient dédiés à leur application. Il est apparu qu'ils avaient clairement des aspects communs, et que l'on pouvait espérer factoriser les efforts de développement dans un unique système. Ce fut le but du projet Descartes.

4.2 Le Système DESCARTES

Le système DESCARTES est un **système général**, c'est à dire non dédié à un domaine d'application spécifique et **entièrement automatique**, c'est à dire ne requérant aucune intervention de l'utilisateur.

Le langage de spécification est un l**angage de type mathématique** et repose sur les notions d'ensemble et de fonctions.

Le langage ne contient ni variables, ni pointeurs, ni structures de données concrètes, ni affectations, ni boucles,.... Les traitements ne sont pas décrits en termes de séquencement dans le temps d'opérations élémentaires. Il y a d'une part description intemporelle de" connaissances ", par le biais de

Exemple de Spécification Descartes

1/ L'application

Considérons l'exemple classique des cours dans une université: Des professeurs donnent 1 à n cours, chaque cours étant dispensé par 1 et 1 seul professeur et suivi par 1 à n étudiants. Chaque étudiant suit 1 à n cours obtenant pour chacun de ces cours 1 et 1 seule note.

2/ Le problème

On s'intéresse à la note maximale obtenue par un étudiant avec un professeur donné, tous cours confondus, et en particulier, on veut écrire un programme qui calcule l'ensemble des notes maximales obtenues par un étudiant avec l'ensemble des professeurs qui sont " sévères ". Admettons par exemple qu'un professeur est sévère s'il existe au moins un cours, parmi les cours qu'il dispense, tel qu'aucun des élèves inscrits au cours n'a eu la moyenne à l'examen.

3/ La spécification Descartes

SOIT EnsembleDesNotesMaxObtenuesAvecLesProfsSévères(e): Etudiants -> P(REELS):
/ ensemble des notes maximales obtenues par un étudiant avec tous les professeurs p tels que p est sévère*/*
{ NoteMaxObtenueAvecUnProf (e, p) | Sévère(p) }

SOIT NoteMaxObtenueAvecUnProf(e, p): Etudiants x Professeurs -> REELS:
/ max des notes d'un étudiant à tous les cours c dispensés par un professeur donné auxquels il est inscrit */*
MAX(Note(e,c) | c ∈ CoursDispensés(p) et e ∈ EtudiantsInscrits(c))

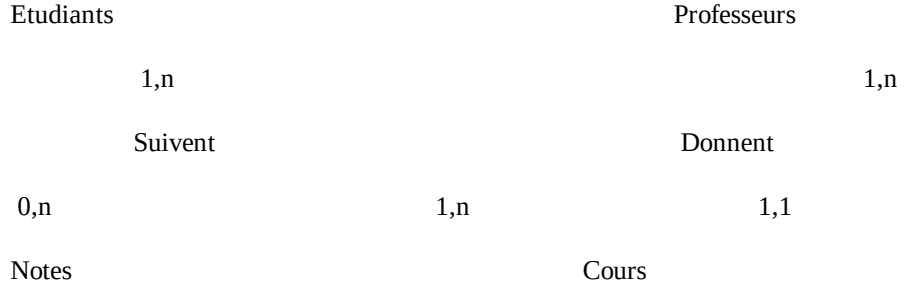
SOIT Sévère(p): Professeurs -> BOOLEEN:
*/*il existe un cours dispensé par le professeur p auquel aucun des étudiants inscrits n'a eu la moyenne */*
∃ c (c ∈ CoursDispensés(p) et ∀ e (e ∈ EtudiantsInscrits(c) => Note(e , c) < 10))

CALCULER EnsembleDesNotesMaxObtenuesAvecLesProfsSévères(e1)

4/ Commentaire

Attention, à la différence des fonctions que l'on écrit dans les langages fonctionnels, **ces fonctions ne sont pas des programmes**. Il n'y a aucune considération informatique. Chaque fonction exprime une connaissance, un concept, indépendamment de toute utilisation particulière. Ces connaissances sont par la suite utilisées par le générateur de programme dès lors que l'on a spécifié l'ordre de calcul.
On calcule l'ensemble des notes maximales obtenues par un étudiant particulier avec l'ensemble de ses professeurs sévères, mais on pourrait demander la même chose pour un ensemble d'étudiants satisfaisant une condition donnée, ou simplement demander si un professeur donné (ou un ensemble de professeurs satisfaisant une condition donnée) est sévère ou non, ou l'ensemble de tous les professeurs sévères,.... Le calcul, s'il est nécessaire dans un contexte particulier, sera chaque fois adapté, c'est à dire optimisé par rapport à ce contexte.

Les fonctions " CoursDispensés ", " EtudiantsInscrits ", et " Note ", à la différence des fonctions précédentes définies par l'utilisateur, sont des fonctions qui font référence aux données de l'application. Elle proviennent directement d'une réécriture sous forme fonctionnelle des associations du schéma conceptuel des données de l'application. De même les ensembles utilisés : " Etudiants ", " Professeurs ",... proviennent en droite ligne des ensembles entités (et éventuellement des ensembles associations) de ce schéma conceptuel des données, schéma qui par ailleurs est exprimé en Descartes dans le modèle Entité-Association usuel.



définitions mathématiques de fonctions et d'autre part une description du résultat désiré par l'intermédiaire d'un ordre de calcul qui indique quelle est la fonction à calculer et sur quel domaine. Il y a aussi description dans le même type de langage fonctionnel et ensembliste des données de l'application.
Plus précisément on peut écrire des formules en logique des prédicats du premier ordre, des définitions de fonctions (éventuellement récursives), ainsi que des définitions d'ensembles. Notons que toutes les primitives usuelles: opérateurs ensemblistes (®, ©, ...), quantificateurs (∀, ∃) ou agrégats (_, _, MAX, ...) qui accompagnent généralement les ensembles sont prédéfinies dans le langage.

A partir d'une telle spécification, le système génère un programme dans un langage de programmation classique, par exemple C. La méthode mise en oeuvre par le système est la **méthode transformationnelle**, ce qui signifie, comme nous l'avons vu précédemment, que la spécification est raffinée progressivement par une série de transformations successives (toutes valides) jusqu'à obtention d'un programme concret exprimé dans un langage de programmation cible qui peut être a priori quelconque.

Une version prototype de DESCARTES a été expérimentée dans le cadre du projet européen DARTS et a permis de générer automatiquement à partir de 600 lignes de spécification formelle un programme PASCAL de 12000 lignes qui a passé avec succès les jeux d'essais officiels du projet en respectant les contraintes "temps réel" imposées.

Il y a actuellement deux versions du système Descartes, l'une écrite dans le langage Descartes lui-même, et l'autre écrite dans le langage Shal, langage permettant de décrire à la fois des buts et des expertises sous formes de règles d'inférences. La version écrite en Shal a permis de générer automatiquement, à l'aide d'un petit générateur provisoire, une version du générateur Descartes écrite en C qui devrait permettre de bootstrapper le système. Cette version écrite en C fait environ 500 000 lignes de code.