Un système d'aide à l'enseignement d'une méthode de programmation

Marc Derroitte, Baudoin Le Charlier

UN SYSTÈME D'AIDE À L'ENSEIGNEMENT D'UNE MÉTHODE DE PROGRAMMATION

Marc Derroitte et Baudouin Le Charlier

Institut d'Informatique, Facultés Universitaires de Namur
Rue Grandgagnage, 21, B-5000 NAMUR (Belgium)
Tel (32) (81) 22 90 65

(e-mail mdr@fun-cs.uucp)

RÉSUMÉ

Cet article présente les objectifs et les concepts d'un système d'aide à l'apprentissage d'une méthodologie de construction de programmes. Ce système est en cours de réalisation à l'Institut d'Informatique des Facultés Universitaires de Namur où cette méthodologie est enseignée. Il permettra d'aider les étudiants à comprendre et à appliquer cette méthode.

La méthodologie se base sur la construction par invariant, c'est-à-dire la description d'une situation générale et la construction de suites d'instructions en fonction de cette situation générale. Les difficultés que peuvent rencontrer ceux qui utilisent la méthode concernent essentiellement le caractère peu intuitif du raisonnement en termes de situation, ainsi que la rigueur nécessaire à la formalisation mathématique des situations. Le système devrait permettre de résoudre ces difficultés.

Le système d'aide à l'apprentissage de cette méthode est basé d'une part sur un langage graphique de description de situations dont le concept fondamental est la notion de "segment", et d'autre part sur un langage mathématique d'expression d'assertions, lui-même basé sur les notions de "suite" et d'"ensemble". Parmi les fonctionnalités du système, mentionnons notamment la vérification par démonstration formelle, la recherche de contre-exemples, des vérifications de cohérence entre les deux types d'expression de situations, des vérifications de cohérence et de complétude par rapport à la méthode.

Dans cet article, nous illustrons également différents scénarios d'utilisation du système, ainsi que le type de remarques qu'il peut offrir. Le problème du "segment de somme maximale" est résolu complètement, selon la méthode supportée par le système.

UN SYSTÈME D'AIDE À L'ENSEIGNEMENT D'UNE MÉTHODE DE PROGRAMMATION

Marc Derroitte et Baudouin Le Charlier

Institut d'Informatique, Facultés Universitaires de Namur
Rue Grandgagnage, 21, B-5000 NAMUR (Belgium)
Tel (32) (81) 22 90 65

(e-mail mdr@fun-cs.uucp)

Mai 1988

1. INTRODUCTION

Il est généralement admis, actuellement, qu'un bon enseignement de la programmation doit reposer sur une méthode rigoureuse de construction de programmes, telle que proposée par J. Arsac [Arsac80][Arsac83], E. Dijkstra [Dijkstra76], D. Gries [Gries81], ... Ces méthodes, bien que les plus intéressantes d'un point de vue théorique, induisent cependant certaines difficultés d'enseignement. En effet, elles exigent de la part des étudiants une formation mathématique adéquate et correspondent peu à une approche intuitive. D'autre part, il existe une tendance actuelle à utiliser l'ordinateur et ses possibilités propres pour compléter bon nombre d'enseignements.

Notre objectif est donc d'utiliser l'ordinateur pour illustrer et supporter ces méthodes d'enseignement de la programmation et pour tenter de les rendre plus accessibles.

Idéalement l'ordinateur devrait permettre de montrer aux étudiants résolvant des problèmes de programmation ce qu'ils ne pourraient percevoir directement, si ce n'est en disposant d'un enseignant prêt à les orienter et à les conseiller au fur et à mesure de l'application pratique d'une de ces méthodes. Le système que nous présentons permet de *comprendre* une telle méthode de construction de programmes par la mise en évidence des éventuelles erreurs de raisonnement.

Il existe des systèmes du genre basés sur la détection et la présentation des erreurs de programmation [Johnson86]. Pour notre part, nous voudrions aborder le problème différemment, en élaborant un système capable d'assister l'étudiant à tous les stades de l'élaboration d'un programme, depuis la spécification jusqu'au programme final.

Dans cet article, nous préciserons d'abord nos objectifs pédagogiques et la méthode enseignée illustrée par un exemple. Nous présenterons ensuite un aperçu du système, puis nous développerons plus en détail les concepts les plus originaux : un langage de description de schémas associé à un langage mathématique d'expression d'assertions. Enfin, nous exposerons les différents scénarios possibles d'utilisation du système en les commentant sur des exemples.

2. MÉTHODE ENSEIGNÉE

2.1. Cadre d'application et options pédagogiques

Comme nous l'avons signalé ci-dessus, la méthode de construction de programmes sur laquelle se basera le système s'apparente aux méthodes classiques de construction de programmes (Arsac, Dijkstra, Gries, ... voir aussi [LeCharlier85], [Leroy78]).

Cependant, ces méthodes ont été adaptées en vue d'objectifs pédagogiques précis. Le système que nous allons présenter devra supporter un cours d'introduction à la programmation donné en première année d'université. Le cours poursuit deux objectifs principaux:

1. faire comprendre à travers l'étude d'un sous-ensemble réduit du langage Pascal [Wirth75] la nature formelle et mécanique des langages de programmation excluant toute imprécision et toute approximation,
2. apprendre une méthode rigoureuse de raisonnement permettant de construire un programme sur des bases solides.

Ces objectifs nous ont conduit à limiter de manière radicale la classe des problèmes traités. Chaque "programme" à construire sera un "morceau" de programme Pascal comportant au plus une boucle et manipulant uniquement des constantes, des variables simples et des tableaux à une dimension. Il sera spécifié de manière précise à la fois pour pouvoir être construit de manière rigoureuse et pour pouvoir servir de primitive dans la construction d'un autre programme. Un programme "compliqué" sera donc systématiquement décomposé en un ensemble de programmes "simples".

C'est à dessein que sont éliminés les problèmes d'entrées/sorties car ceux-ci sont plus ardus à spécifier de manière simple et rigoureuse en raison de leur caractère dynamique. D'autre part, les problèmes traités sont purement "techniques" sans référence aux applications "réelles". C'est aussi une option délibérée car le traitement rigoureux de problèmes "pratiques" exige une modélisation du réel qui augmente considérablement les difficultés d'une approche rigoureuse. Dans ce cours d'introduction, nous voulons tenter de motiver les étudiants par la démarche elle-même, conduisant directement à un programme correct, pour des raisons claires et logiques. Notre optique risque peut-être de frustrer

l'étudiant en le privant de l'impression d'avoir réalisé un programme utile qui "tourne", mais nous estimons lui donner de la sorte de meilleures armes pour s'attaquer ensuite à des problèmes plus gros et plus "réels" exigeant un aspect de modélisation plus important. Car il saura déjà ce que veut dire qu'un programme est correct et comment le construire.

Le système présenté dans cet article devrait aussi, nous l'espérons, motiver l'étudiant à un autre niveau en l'accompagnant dans sa démarche de construction et en l'aidant à valider ses raisonnements.

2.2. Démarche proprement dite

A partir d'un énoncé déjà précis, en français, et éventuellement d'une idée intuitive de solution, les étapes décrites ci-dessous doivent être suivies.

2.2.1. Spécification

La spécification du programme, sur base de l'énoncé proposé, comporte trois parties :

1. La *liste des objets utilisés* (constantes, variables, tableaux) classés en :
   - objets principaux qui constituent les données et résultats du programme,
   - objets auxiliaires qui constituent principalement les variables de travail[1].

2. La *précondition* du programme (notée *Pré*) ou *situation initiale* qui est l'ensemble des conditions que doivent respecter les valeurs des objets principaux pour que l'exécution du programme ait un sens.

3. La *postcondition* du programme (notée *Post*) ou *situation finale* qui est la description de l'état des objets principaux après l'exécution du programme (si la précondition était respectée avant l'exécution).

2.2.2. Situation générale et Invariant

Pour les programmes ayant une forme itérative, la situation générale est une description schématique de l'ensemble des conditions vérifiées par les objets utilisés à chaque itération.

---

(1) Il peut paraître contraire à la notion de spécification d'y inclure la liste des objets auxiliaires. Celle-ci est cependant nécessaire si le programme est utilisé comme sous-problème dans la construction d'un autre programme, pour vérifier qu'un objet auxiliaire du sous-problème n'est pas utilisé par le programme "principal". En effet, un sous-problème ne constituant pas une procédure mais seulement un morceau de programme, il n'y a pas de notion de variable locale. Cette partie de la spécification n'est pas remplie a priori, mais au cours de l'élaboration du programme.

On appellera Invariant (noté I_A) une situation générale décrite dans un langage mathématique.

2.2.3. Choix des instructions

Sur base de la situation générale, plusieurs suites d'instructions doivent être décrites. Chacune de ces suites d'instructions doit vérifier une condition du type {P} S {Q}, où P et Q sont des situations (ou assertions), S la suite d'instructions considérée, et qui signifie :

"Si P est vrai avant l'exécution de S, alors l'exécution de S se terminera et les valeurs finales des objets du programme vérifieront Q".

Ces suites d'instructions sont :

- l'initialisation (notée *Init*) qui est l'ensemble des instructions nécessaires à la vérification de la condition :

  {Pré} Init {Ia}.

- la condition de terminaison (notée *B*) de l'itération et la clôture (notée *Clôt*) qui est l'ensemble des instructions nécessaires à la vérification de la condition :

  {Ia et B} Clôt {Post}.

- l'itération (notée *Iter*) qui est l'ensemble des instructions nécessaires à la vérification de la condition :

  {Ia et non(B)} Iter {Ia}.

2.2.4. Vérification de la terminaison

La vérification de la terminaison se fait par la définition d'une fonction entière et bornée des valeurs des variables, qui croît strictement à chaque exécution de la suite d'instructions Iter.

2.2.5. Ecriture du programme

Toutes les suites d'instructions doivent être assemblées de manière à former un seul programme (ou morceau de programme) respectant la syntaxe du langage Pascal.

2.3. Exemple

Un problème entrant dans la catégorie des problèmes "simples" que nous prenons en considération est celui de la recherche dans un tableau du segment de somme

maximale. Ce problème classique a notamment été proposé par Jacques Arsac dans son article intitulé "La conception des programmes" publié dans Pour la science [Arsac84].

Voici une solution complète de ce problème selon notre démarche. Le lecteur qui ne serait pas familiarisé avec une méthode de construction de programmes de ce type veillera à étudier soigneusement cet exemple en vue d'une compréhension plus aisée de ceux qui seront traités dans la suite.

2.3.1. Enoncé du problème

Soit a, un tableau initialisé de n éléments de type entier. Ces éléments, notés a[i] avec $1 \leq i \leq n$, sont positifs, négatifs ou nuls. Ecrire un programme pour déterminer les indices d et f tels que la somme des valeurs des éléments du segment a[d..f] soit maximale.

Contrainte : tout élément du tableau ne peut être consulté qu'une seule fois.

Convention : la somme des valeurs des éléments d'un segment vide est nulle.

2.3.2. Solution

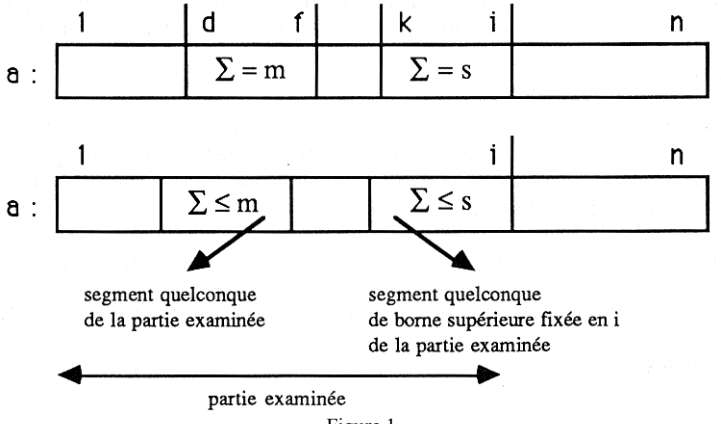
2.3.2.1. Spécification

- Objets utilisés :
  - objets principaux : a : array[1..n] of integer;
$n \geq 0$, constante;
d, f : integer;
  - objets auxiliaires : i, s, m, k : integer;
- Précondition :
  - a[1..n] initialisé;
- Postcondition :
  - $1 \leq d \leq f+1 \leq n+1$
  - $\forall\, i, j : 1 \leq i \leq j+1 \leq n+1 : \sum_{k=i}^{j} a[k] \leq \sum_{k=d}^{f} a[k]$
  - a inchangé

2.3.2.2. Description graphique de la situation générale

Supposons que l'on a déjà effectué une partie du travail : on a déjà examiné la partie a[1..i] du tableau. Pour espérer posséder suffisamment

d'informations en fin d'exécution pour résoudre le problème posé, il faut que la situation suivante soit établie :



- Figure 1 -

2.3.2.3. Description mathématique de la situation générale

Invariant :

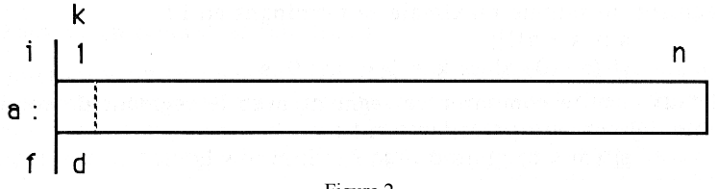
- $1 \le k \le i+1 \le n+1$
- $1 \le d \le f+1 \le i+1$
- $\sum_{p=d}^{f} a[p] = m$
- $\sum_{p=k}^{i} a[p] = s$
- $\forall\ d', f' : 1 \le d' \le f'+1 \le i+1 : \sum_{p=d'}^{f'} a[p] \le m$
- $\forall\ k' : 1 \le k' \le i+1 : \sum_{p=k'}^{i} a[p] \le s$

2.3.2.4. Choix des instructions

Le choix des instructions est fait ici en raisonnant sur la situation représentée graphiquement.

- Initialisations :

_ Pour établir la situation générale de la façon la plus simple, il suffit d'établir la situation suivante :



- Figure 2 -

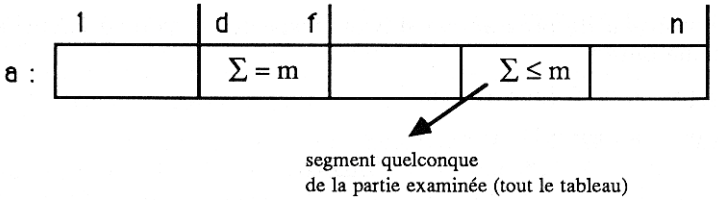
En effet, si i = 0, le seul segment possible inclus dans a[1..i] est le segment vide dont la somme des éléments est nulle.

_ La suite d'instructions d'initialisation sera donc :

d := 1; f := 0; m := 0;
k := 1; i := 0; s := 0;

- Condition de terminaison et clôture :

_ La situation constituant un cas particulier de la situation générale, mais la plus proche de la situation finale, est la suivante :



- Figure 3 -

En effet, cette situation est vérifiée lorsque i = n, donc lorsque tout le tableau a été examiné.

D'autre part, on constate que la Postcondition est exactement la description de cette situation, donc qu'il n'y aura nul besoin d'instructions de clôture.

_ La condition de terminaison sera :

(i = n)

et la suite des instructions de clôture sera vide.

- Itération :

  - Si la condition de terminaison n'est pas vérifiée, pour s'approcher de la situation finale, il est nécessaire d'augmenter la partie du tableau déjà examinée.

    Pour cela, nous aurons l'instruction :

    i := i + 1;

    Pour rétablir la situation générale, il faut d'abord déterminer le segment de somme maximale se terminant en i :

    s := s + a[i];
    si (s ≤ 0) alors k := i+1; s := 0 is

    Il faut ensuite comparer ce segment avec le segment de somme maximale déjà obtenu précédemment :

    si (m < s) alors d := k; f := i; m := s is

  - La suite d'instructions d'itération rétablissant la situation générale sera donc :

```
i := i + 1;
s := s + a[i];
si (s ≤ 0) alors k := i+1; s := 0 is
si (m < s) alors d := k; f := i; m := s is
```

2.3.2.5. Vérification de la terminaison

La valeur de la variable i est bornée par n et croît strictement à chaque itération.

2.3.2.6. Ecriture du programme

Le morceau de programme suivant respecte la syntaxe Pascal et résout le problème posé :

```
const   n = 10 {par exemple};

var     a : array[1..n] of integer;
            i, k, s, d, f, m : integer;

begin
        d := 1; f := 0; m := 0;
        k := 1; i := 0; s := 0;
        while (i <> n) do
        begin
                i := i + 1; s := s + a[i];
                if (s ≤ 0) then begin k := i + 1; s := 0 end;
                if (m < s) then begin m := s; d := k; f := i end
        end
end
```

2.4. Problèmes rencontrés

La méthode de construction de programmes telle que nous venons de la décrire est effectivement enseignée à l'Institut d'Informatique de Namur. Cet enseignement nous permet de cerner précisément les problèmes rencontrés par les étudiants dans l'application de la méthode.

Il semble possible de répartir de manière générale ces problèmes selon deux catégories que nous allons brièvement décrire.

2.4.1. Raisonner en termes de "situations"

La puissance de ce mode de construction de programmes repose sur le fait qu'il permet de structurer ceux-ci, non pas en termes de séquences d'instructions, mais en termes de *situations successives*. Il est ainsi possible de raisonner sur le nombre limité d'opérations qui permet de "passer" d'une situation à l'autre, plutôt que directement sur la globalité du problème, ce qui, même pour la classe des problèmes envisagée, dépasse bien souvent les capacités de l'esprit humain.

Malheureusement, cette manière de raisonner n'est pas intuitive. Les étudiants préfèrent, naturellement, simuler le déroulement d'un exemple et construire directement le programme, même s'ils sont contraints de tenter par la suite de justifier le résultat par la méthode proposée pour "contenter" l'enseignant.

2.4.2. Formulation "mathématique" des énoncés

La démarche proposée peut être appliquée avec deux niveaux de rigueur. Le premier permettant simplement de construire correctement le programme correspondant à une spécification, le second impliquant en plus de *démontrer* la correction de ce programme en justifiant de façon rigoureuse les différentes étapes de la démarche.

Dans la pratique, on constate une réelle difficulté de la part des étudiants pour franchir le cap du premier niveau. En effet, étant donné la classe des problèmes envisagés, il est presque toujours possible d'exprimer une situation sous la forme d'un schéma (voir exemple ci-dessus) assez simple et correspondant bien à une vue intuitive des choses. Cependant, même si l'étudiant s'acquitte correctement de cette tâche, il parvient généralement avec beaucoup plus de peines à fournir une description mathématique correcte de la même situation.

Il est alors en mesure de construire une solution correcte d'après la description "schématique", mais ne peut justifier rigoureusement son raisonnement.

3. APERÇU DU SYSTÈME

L'outil que nous envisageons de réaliser, en tant qu'environnement d'aide à l'utilisateur, devrait apporter une solution concrète aux deux types de difficultés présentés ci-dessus.

En effet, le système sera doté de deux langages d'expression de spécifications et de situations :

- un langage graphique de représentation de schémas, basé sur la notion de "segment",
- un langage mathématique d'expression d'assertions, basé sur les notions de "suite" et d'"ensemble".

Le système disposera aussi de plusieurs fonctions de vérification :

- vérifications syntaxiques,
- vérification de la cohérence entre l'expression graphique et mathématique d'une même situation,
- vérification de la correction d'une séquence d'instructions par rapport à une situation initiale et une situation finale,
- vérifications de cohérence entre différentes étapes de la démarche,
- vérifications de complétude,
- ...

En fonction de toutes ces vérifications, le système pourra fournir à l'utilisateur, à chaque étape, des commentaires sur la manière dont il a réussi cette étape, allant du simple signal d'erreur à l'explication du problème de raisonnement qui a provoqué l'erreur, en passant par la présentation de contre-exemples.

Enfin, le système devra proposer plusieurs scénarios d'utilisation en fonction des besoins :

- un scénario standard consistant à suivre une à une toutes les étapes de la démarche, avec la possibilité de retours en arrière en fonction des commentaires du système,
- la possibilité d'effectuer isolément une étape particulière de la démarche,
- un scénario basé uniquement sur le langage graphique ou uniquement sur le langage mathématique,
- ...

Ainsi conçu, nous espérons que notre outil :

- aidera les étudiants à réfléchir sur la manière dont ils résolvent un problème, et à raisonner avec une plus grande précision, en utilisant le langage mathématique,
- leur apportera une assistance proche de celle qui pourrait leur être fournie par un instructeur, dès qu'elle s'avèrera nécessaire (plutôt qu'en fin d'exercice),
- leur fournira un environnement agréable, les engageant à utiliser la méthode telle que nous la proposons.

Les principales fonctionnalités du système tel que nous envisageons de le construire sont décrites plus en détail dans la suite de cet article.

4. LANGAGE DE DESCRIPTION DE SCHÉMAS

Dans la plupart des cas, les situations à considérer peuvent être décrites sous forme d'un *ensemble de schémas* représentant chacun un contenu de tableau. Chaque schéma peut se caractériser par un *ensemble de segments contigus*, des conditions à vérifier par le contenu des segments et des relations entre ces segments.

Notre système mettra donc à la disposition de l'utilisateur un outil de description et de manipulation de schémas dont le concept fondamental sera la notion de segment.

Nous décrivons ci-dessous les principaux concepts et mécanismes de ce langage.

4.1. Le concept de segment

Un segment est une suite d'éléments consécutifs d'un tableau.

Il peut se caractériser par :

- la manière dont ses bornes sont fixées,
- les propriétés de son contenu,
- ses liens avec d'autres segments.

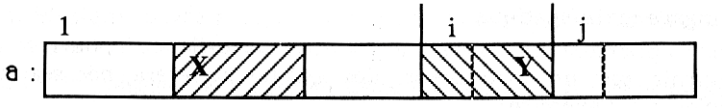
4.1.1. Règles relatives aux bornes des segments

- Un segment est repéré par ses bornes.

  La *borne* inférieure (supérieure) d'un segment est l'indice de son premier (dernier) élément.

  Une borne est fixe ou mobile. Elle est *fixe* si elle est liée à la valeur d'une variable c'est-à-dire si la valeur de cette variable est en permanence égale à la borne (ou, éventuellement, à l'indice précédant ou suivant la borne). Une borne est *mobile* si elle n'est pas liée à la valeur d'une variable. Sa valeur est alors quelconque.

Exemple :



- Figure 4 -

Les bornes du segment X sont mobiles et celles du segment Y sont fixes puisque sa borne inférieure vaut i et sa borne supérieure vaut j-1.

- Un *segment* est *fixe* ou *mobile*. Il est fixe si ses deux bornes sont fixes. Il est mobile si au moins une de ses bornes est mobile. Un segment mobile représente en fait une famille de segments : tous les segments possibles dans l'espace de mobilité des bornes.

  Exemple : le segment X est mobile : il représente tous les segments possibles de a[1..i-1].

- Un segment peut être *nommé*. Dans cet article, nous nommerons les segments par des lettres majuscules (X, Y, ...).
- Un segment peut être *vide*. Il est vide si sa borne inférieure est égale à sa borne supérieure plus un.

4.1.2. Propriétés du contenu des segments

Le langage graphique devra permettre de décrire un grand nombre de propriétés du contenu d'un segment.

Pour cela, il faudra qu'il dispose d'un ensemble très large de primitives de sorte que l'utilisateur puisse se contenter de les combiner de manière simple sans rencontrer de problèmes de formulation mathématique et qu'ainsi la définition de schémas garde au maximum son caractère intuitif. Idéalement, le système devrait permettre de réaliser des schémas aussi simples à comprendre que ceux que l'on construirait de manière ad hoc, pour un problème déterminé.

Les propriétés du contenu d'un segment s'exprimeront par des conditions à indiquer dans la représentation graphique du segment. Une condition pourra comporter des éléments tels que constantes, variables, opérateurs arithmétiques et logiques, mais aussi des primitives plus puissantes et des notations simplificatrices dont voici quelques exemples :

- un élément quelconque d'un segment sera noté "■",
- l'indice d'un élément quelconque sera noté "i(■)",

Ainsi :

- " ■ ≤ 0 " signifie tous les éléments du segment sont négatifs ou nuls.
- " ■ pair " signifie tous les éléments du segment sont pairs
- " S = s " signifie la somme des éléments du segment est égale à la valeur de la variable s.
- " croissant " signifie les éléments du segment sont triés de façon croissante.
- " permuté " signifie le contenu du segment constitue une permutation de son contenu initial.

On peut également admettre que l'information que l'on veut exprimer décrit incomplètement une situation précise, ce qui permettra d'envisager des propriétés plus "floues" telles que la suivante :

- " examiné " signifie l'algorithme a déjà passé en revue le contenu du segment.

Même si cette condition est imprécise, elle permet certaines vérifications.

4.1.3. Liens entre les segments

Pour exprimer au mieux l'ensemble des situations rencontrées dans la classe des problèmes traités, le langage devra aussi offrir la possibilité de décrire des relations entre segments.

Certaines de ces relations peuvent s'exprimer par la présence des segments sur un même schéma :

- Deux segments quelconques figurant sur un même schéma sont disjoints.
- Si deux segments sont contigus (sur le schéma), la borne supérieure du premier est égale à la borne inférieure du second moins un.
- Si un segment est avant un autre (sur le schéma), sa borne supérieure est inférieure à la borne inférieure de l'autre.

D'autres relations doivent s'exprimer en représentant les segments sur des schémas différents. Par exemple, pour exprimer que X et Y sont totalement indépendants, il suffit de les représenter sur des schémas différents.

Exemples de relations :

- " ■ ∈ Y " signifie l'ensemble des éléments du segment contenant cette propriété est inclus dans l'ensemble des éléments du segment nommé Y.
- " Σ ≤ Σ (Y) " signifie la somme des éléments du segment est inférieure ou égale à la somme des éléments du segment Y.

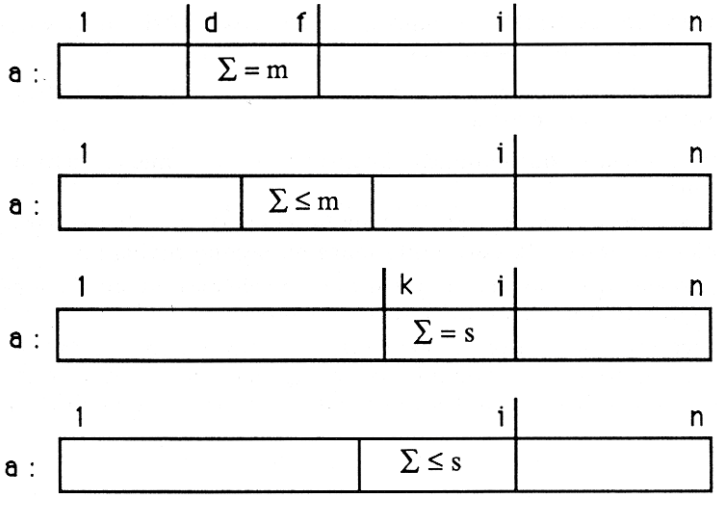
Et des relations plus compliquées peuvent être exprimées en imposant à deux segments des conditions dans lesquelles figurent des variables communes.

4.2. Application de ces concepts à la description de situations

Pour décrire une situation en utilisant les concepts définis ci-dessus, il sera souvent nécessaire de superposer plusieurs schémas représentant des situations compatibles.

Le système vérifiera cette compatibilité (si c'est possible) et affichera éventuellement un schéma de synthèse résumant la situation, mais ne respectant pas les règles d'interprétation stricte des schémas.

Par exemple, la description de la situation générale de l'exercice présenté au paragraphe 2.3 devra se faire à l'aide des quatre schémas suivants :



- Figure 5 -

Interprétation :

_ schéma 1 :

- le segment [d..f] est fixe et la somme de ses éléments est égale à m,
- ce segment est inclus dans la partie [1..i] du tableau a,
- ce segment peut être vide.

_ schéma 2 :

- la somme des éléments du segment mobile représenté est inférieure ou égale à m,
- donc, la somme des éléments de tous les segments possibles de la partie [1..i] du tableau a est inférieure ou égale à m,
- donc, le segment [d..f] est le segment de somme maximale de la partie [1..i] du tableau a.

_ schéma 3 :

- le segment [k..i] est fixe et la somme de ses éléments est égale à s,
- ce segment peut être vide.

_ schéma 4 :

- la somme des éléments du segment mobile dont la borne supérieure est fixe (liée à la variable i) est inférieure ou égale à s,
- donc, la somme des éléments de tous les segments possibles dont la borne supérieure est i est inférieure ou égale à s,
- donc, le segment [k..i] est le segment de somme maximale dont la borne supérieure est i.

Ces quatre schémas sont superposables : ils représentent des situations compatibles.

Le système pourrait effectuer des superpositions pour afficher la même situation grâce aux deux schémas tels que présentés au paragraphe 2.3 (figure 1). Mais il faut noter que s'ils étaient directement introduits par l'utilisateur, les deux ensembles de schémas ne représenteraient pas la même situation et donneraient lieu à des interprétations différentes par le système.

En effet, face à la situation correspondant aux schémas de la figure 1, le système effectuerait notamment les interprétations suivantes :

- le segment [d..f] est avant le segment [k..i],
- $f < k$,
- le segment mobile caractérisé par " $\Sigma \leq m$ " est avant le segment mobile caractérisé par " $\Sigma \leq s$ ".

Ces interprétations ne correspondent ni à l'interprétation intuitive de la situation générale, ni à l'interprétation par le système de la situation représentée par l'ensemble des quatre schémas ci-dessus.

5. LANGAGE MATHÉMATIQUE

5.1. Concepts

Pour exprimer de manière mathématique les situations auxquelles nous nous intéressons, il paraît suffisant de pouvoir construire des formules manipulant des quantificateurs et permettant de créer et manipuler des ensembles et des suites. Un langage de ce type est défini dans [Fisette85] [Wenzi87].

Le langage devra donc disposer d'une large gamme d'opérateurs puissants et de mécanismes de composition de ceux-ci. La syntaxe devra être aussi agréable que possible grâce à l'utilisation des symboles mathématiques habituels et d'opérateurs infixés.

Ainsi, l'invariant et les spécifications présentées dans l'exercice du segment de somme maximale pourront pratiquement être écrits tels quels dans notre langage.

5.1.1. Notions de suite et d'ensemble

De même que dans le langage graphique la notion de base était le segment, ce langage-ci peut être construit sur base des notions de suite et d'ensemble. Ces deux notions étant couramment utilisées en informatique, nous n'en donnerons pas ici une définition formelle. Une connaissance intuitive de ces notions est en effet suffisante pour la compréhension de cet article.

Par exemple, si, dans un schéma représentant le tableau a, un segment fixe dont les indices des bornes sont 1 et iest caractérisé par la propriété "# = s" (signifiant que le nombre d'éléments de valeurs différentes du segment est égal à la valeur de la variable s), alors, dans le langage mathématique, cette propriété pourra s'exprimer par "#{a[1..i]}=s" où les accolades servent à désigner l'ensemble des valeurs contenues dans a[1..i].

5.1.2. Opérateurs

Les opérateurs qui devront être mis à la disposition de l'utilisateur devront être suffisamment puissants pour que l'expression d'une situation ne constitue pas un effort important de programmation.

Exemples d'opérateurs : $\forall$, $\exists$, Σ, $\prod$, $\cap$, $\supset$, max, crois, ...

5.1.3. Définition de nouvelles primitives

Malgré la dotation de primitives puissantes, pour certains problèmes, le langage ne pourra que difficilement exprimer les situations relatives à ceux-ci. Il faudra donc prévoir des mécanismes permettant à l'utilisateur de définir de nouvelles primitives.

Cependant, la définition de ces primitives posera souvent des problèmes de programmation plus difficiles que ceux de la classe envisagée au départ. Cette tâche sera donc réservée aux enseignants désireux d'accroître la puissance d'expression du langage.

5.2. Problèmes de syntaxe

Comme nous l'avons dit ci-dessus, au point de vue de la syntaxe du langage mathématique, notre objectif est que celle-ci soit la plus proche de celle utilisée généralement sur papier, et donc la plus agréable possible pour l'utilisateur.

Cependant, cette syntaxe est impossible à mettre en oeuvre étant donné les limites liées à l'implémentation et aux outils utilisés (un clavier ne possède pas l'ensemble des caractères que nous voudrions voir apparaître dans la syntaxe). La solution que nous avons retenue est de définir deux niveaux de langage, le premier concernant l'introduction des expressions mathématiques et le second concernant leur affichage.

Ces remarques sont également valables, quoique dans une moindre mesure, pour le langage de description de schémas, nous adopterons donc pour celui-ci une solution du même type.

6. SCÉNARIOS D'UTILISATION DU SYSTÈME

Dans cette partie, nous décrirons l'interface entre l'utilisateur et le système : quels services pourront être offerts par le système, quels choix seront laissés à l'utilisateur.

Signalons dès à présent la caractéristique essentielle de tous les scénarios d'utilisation du système : la possibilité permanente pour l'utilisateur d'effectuer autant de retours en arrière qu'il le souhaite. Cette possibilité est rendue indispensable par le fait que d'une part, une erreur à une étape pourra en provoquer une autre à une étape suivante et que d'autre part, certains types d'erreurs ne pourront pas donner lieu à un diagnostic dans l'étape où elles ont été commises mais plus tard dans la démarche, le système se bornant à faire dans l'étape courante des vérifications de cohérence.

6.1. Scénario standard

Une façon d'utiliser le système sera d'appliquer la méthode telle que nous l'avons décrite, de fournir au système les résultats de chacune des étapes. Ces résultats seront alors

évaluées et l'étape suivante de la démarche sera proposée lorsqu'ils seront jugés satisfaisants.

Imaginons un étudiant utilisant le système et résolvant l'exercice du segment de somme maximale présenté précédemment.

Pour chaque étape de la démarche, nous exposerons ci-dessous :

- des résultats possibles que cet étudiant pourrait fournir au système,
- les réactions du système face à ces résultats,
- des commentaires sur la façon dont le système pourrait s'y prendre pour engendrer ces réactions.

Le lecteur pourra comparer en permanence les résultats proposés avec ceux considérés comme corrects, présentés dans l'exemple.

6.1.1. Construction de la spécification

■ Interventions de l'utilisateur :

L'utilisateur devra d'abord décrire les objets principaux. Ensuite, il décrira la précondition et la postcondition d'une part sous forme de schémas, d'autre part dans le langage formalisé.

Par contre, pour les objets auxiliaires, l'utilisateur ne sera pas tenu d'en fournir la liste à ce moment, il pourra les accumuler au fur et à mesure des étapes suivantes de la démarche, chaque fois qu'ils lui apparaîtront nécessaires.

■ Vérifications du système :

Il est utopique d'espérer une vérification de la correction de spécifications par rapport à un énoncé intuitif puisque ce dernier est non formalisé. On ne pourra donc envisager, à cette étape, que des vérifications de cohérence.

Citons quelques vérifications possibles :

- vérifications syntaxiques par rapport aux langages graphique et mathématique,
- cohérence des schémas (ils expriment bien des situations non contradictoires ou réalisables),
- vérifications de complétude (tous les objets principaux apparaissent dans la précondition ou la postcondition, toutes les variables utilisées dans la précondition et la postcondition sont bien déclarées dans les objets principaux, la précondition ne contient pas d'objets qui ne sont plus référencés dans la postcondition, ...),
- ...

■ Exemples :

1 U:[2] objets principaux :

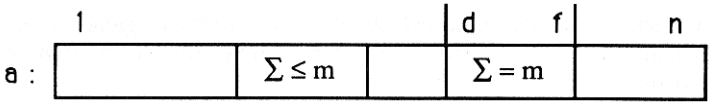
a : array[1..n];
d , f : integer;

S: Les éléments du tableau a sont-ils de type quelconque ?

Le tableau a peut-il être vide ?

2 U: précondition : /

postcondition :



- Figure 6 -

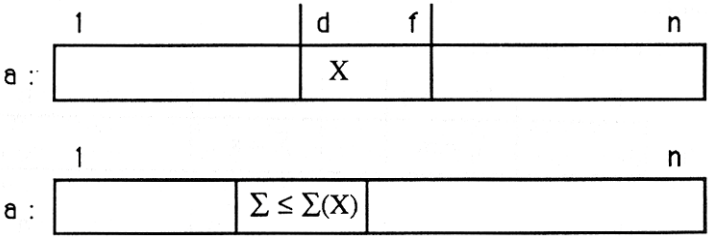
S: 1. Le contenu du tableau ne devrait-il pas être initialisé ?

2. Interprétation du schéma :

- la somme des éléments du segment fixe [d..f] est égale à m,
- le segment mobile dont la somme est inférieure ou égale à m est avant le segment fixe [d..f],
- donc le segment [d..f] est de somme supérieure ou égale à celle de tous les segments qui le précèdent.

C: Nous espérons que la présentation de l'interprétation du schéma par le système permette à l'utilisateur de vérifier si cela correspond bien à son interprétation intuitive. Dans ce cas-ci, l'utilisateur constatera qu'il ne souhaitait pas représenter une telle situation et se rendra compte de la nécessité de deux schémas pour l'exprimer.

3 U: - Formulation graphique :



- Figure 7 -

(2) U = utilisateur, S = système, C = commentaires.

- Formulation mathématique :

- $1 \leq d \leq f+1 \leq n+1$
- $\forall\, i, j : 1 \leq i \leq j+1 \leq n+1 : , \sum_{k=i}^{j} a[k] < \sum_{k=d}^{f} a[k]$

S: O.K., Les deux formulations ne sont pas contradictoires.

6.1.2. Description de la situation générale

■ Interventions de l'utilisateur :

L'utilisateur devra d'abord décrire la situation générale à l'aide du langage graphique, et ensuite fournir un invariant équivalent en langage mathématique.

■ Vérifications du système :

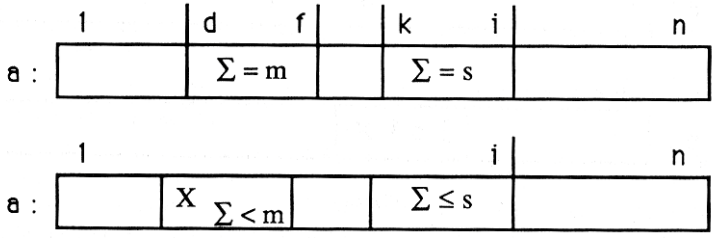
Le système pourra vérifier la cohérence de la description graphique : les schémas représentent des situations possibles, ils sont bien superposables, ... Il fera également le même genre de tests pour l'invariant.

Le système pourra ensuite effectuer des comparaisons et des tests de cohérence entre les deux descriptions. Remarquons cependant que leur équivalence logique ne pourra être démontrée formellement que pour des conditions simples comme le non-dépassement de bornes. Il faudra donc mettre en oeuvre d'autres techniques de vérification telles que des tests sur base d'exemples (le système génère des situations réelles correspondant à la situation générale, en donnant des valeurs aux variables et tableaux, teste si l'invariant est vérifié pour ces situations réelles et, le cas échéant, présente le contre-exemple à l'utilisateur).

Signalons enfin que d'autres vérifications concernant la situation générale pourront être effectuées dans les étapes suivantes.

■ Exemples :

1 U: situation générale :



- Figure 8 -

S: L'interprétation du schéma conduit à une contradiction : le segment X et le segment [d..f] sont caractérisés par des propriétés incompatibles et peuvent coïncider.

C: En effet, le segment X est mobile, il représente donc un ensemble de segments dont une instance pourrait être [d..f], cela signifie donc que la somme des éléments de [d..f] devrait être en même temps strictement inférieure et égale à m !

2 U: situation générale : voir Figure 1 de l'exemple (§ 2.3.2.2),

Invariant : voir invariant de l'exemple (§ 2.3.2.3).

S: Incohérence entre le schéma et sa formulation mathématique : l'interprétation du schéma exprime que f < k, mais cette propriété ne se retrouve pas dans la formulation mathématique.

6.1.3. Choix des instructions

■ Interventions de l'utilisateur :

L'utilisateur devra exprimer successivement les suites d'instructions suivantes: l'initialisation, la condition de terminaison, la clôture et l'itération.

■ Vérifications du système :

Les principales vérifications se feront dans cette étape, grâce à des techniques de recherche de contre-exemples et à l'expertise du système.

Le système vérifiera la syntaxe des suites d'instructions, les propriétés liant chaque suite d'instructions à la situation générale et présentera les erreurs sous forme de schémas, contre-exemples, ...

■ Exemples :

1 U: initialisations : d := 1; f := 1; m := a[1];
k := 1; s := a[i];
i := 2;

S: L'ordre des affectations est incorrect : lors de l'exécution de l'affectation " s := a[i] ", la valeur de i est indéterminée.

De plus, si n = 0, l'élément a[1] n'existe pas et donc l'affectation " m := a[1] " n'a pas de sens.

2 U: condition de terminaison :

i = n+1

S: contradiction avec la condition " i+1 ≤ n+1 " de l'invariant.

C: En effet, " i+1 ≤ n+1 " est équivalent à " i ≤ n ", donc la condition de terminaison ne sera jamais remplie.

3 U: clôture :

m := 0;

S: L'instruction de clôture est inutile puisqu'elle affecte une variable qui n'intervient pas dans la postcondition.

4 U: itération :

... ; i := i+2; ...

S: La condition " i+1 ≤ n+1 "de l'invariant n'est pas vérifiée après l'exécution de l'itération.

C: En effet, une situation permise avant l'itération est " i = n-1 ", donc, après celle-ci, on aura" i = n+1 ", situation non permise par l'invariant.

6.1.4. Vérification de la terminaison

■ Interventions de l'utilisateur :

Expression d'une fonction des valeurs des variables bornée et strictement croissante.

■ Vérifications du système :

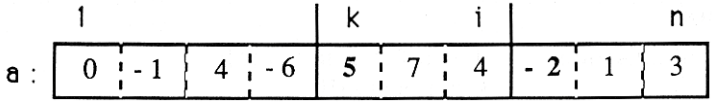
Le système vérifiera si cette fonction est bien bornée et croît bien à chaque exécution de l'itération.

■ Exemples :

1 U: fonction bornée et strictement croissante :

f = k

S: Contre-exemple :



- Figure 9 -

Si on exécute les instructions d'itération dans cette situation, on constate que la valeur de k n'est pas modifiée.

6.1.5. Ecriture du programme

■ Interventions de l'utilisateur :

Regrouper les différentes suites d'instructions et en faire un morceau de programme Pascal.

■ Vérifications du système :

Les principales vérifications que le système pourra effectuer consisteront à vérifier que le programme reprend bien les suites d'instructions décrites précédemment et que la syntaxe Pascal est respectée.

Le système pourra également fournir un environnement de test à l'utilisateur en lui offrant un dialogue d'introduction des données de son programme, en exécutant celui-ci, en vérifiant les situations décrites lors de la construction et en présentant à l'utilisateur les résultats.

6.2. Autres scénarios

D'autres scénarios d'utilisation du système seront proposés :

- suivre la démarche selon le premier niveau de rigueur, c'est-à-dire en n'exprimant les situations que sous forme de schémas, sans utiliser le langage mathématique,
- inversement, ne fournir au système que l'expression mathématique des situations,
- exécuter une ou plusieurs étapes particulières de la démarche, par exemple la dernière pour vérifier qu'un programme quelconque est syntaxiquement correct,
- ...

6.3. Niveaux de vérification

Plusieurs niveaux de vérification peuvent être définis selon la complexité du problème.

1. Vérifications simples de cohérence et complétude avec messages d'erreur.
2. Présentation des erreurs détectées sous forme de contre-exemples.
3. Interprétation de ces erreurs sur base d'une certaine expertise (détection des erreurs de raisonnement classiques).
4. Démonstrations formelles de l'exactitude des raisonnements.

Dans les limites du possible, notre système tentera de fournir à l'utilisateur l'aide de niveau maximum.

7. PERSPECTIVES

Comme nous l'avons mentionné, la réalisation de ce système fait l'objet d'un projet de recherche à l'Institut d'Informatique de Namur.

Outre les points développés dans cet article, cette recherche porte notamment sur la constitution d'un répertoire d'exercices représentatifs abordés par les étudiants, l'analyse de la résolution de ces exercices afin d'étoffer l'expertise à intégrer dans le système, l'étude

des techniques existantes de détection et d'interprétation des erreurs de programmation et des heuristiques de construction d'invariants.

Après implémentation, le système sera expérimenté avec la collaboration d'étudiants. Cette expérimentation devra permettre d'y apporter des améliorations au point de vue interface aussi bien qu'au point de vue efficacité de l'expertise.

8. REMERCIEMENTS

Les auteurs remercient tous les membres de l'Institut d'Informatique qui ont lu et critiqué une version précédente de cet article : G. Barreto, J. Burnay, P. De Boeck, B. Delcourt, Y. Deville, F. Dubisy, J.-P. Hogne, J.-M. Jacquet et G. Thiry.

RÉFÉRENCES

[Arsac80] J. ARSAC, *Premières leçons de programmation* , Nathan, Paris, 1980.

[Arsac83] J. ARSAC, *Les bases de la programmation* , Dunod informatique, Paris, 1983.

[Arsac84] J. ARSAC, *La conception des programmes* , In Pour la science , Novembre 1984.

[Dijkstra76] E.W. DIJKSTRA, *A discipline of programming* , Prentice Hall, 1973.

[Fisette85] D. FISETTE, *Définition d'un langage de programmation permettant l'expression d'assertions* , mémoire de maîtrise, Namur, 1985.

[Gries81] D. GRIES, *The Science of Programming* , Springer Verlag, 1981.

[Johnson86] W.L. JOHNSON, *Intention-Based Diagnosis of Novice Programming Errors* , Pitman, London, 1986.

[LeCharlier85] B. LE CHARLIER, *Réflexions sur le problème de la correction des programmes* , thèse de doctorat, Namur, Janvier 1985.

[Leroy75] H. LEROY, *La fiabilité des programmes* , Presses universitaires de Namur, 1975.

[Leroy78] H. LEROY, *La fiabilité des programmes (2)* , Ecole d'été de l'A.F.C.E.T., Namur, Juillet 1978.

[Wenzi87] M. WENZI, *Implémentation des suites et ensembles pour un langage de programmation permettant l'expression d'assertions* , mémoire de maîtrise, Namur, 1987.

[Wirth75] N. WIRTH, K. JENSEN, *Pascal User Manual and Report* , Springer Verlag, New York, 1975.