MASTER EN SCIENCES INFORMATIQUES

Un logiciel d'aide à l'enseignement d'une méthode de programmation: étude conceptuelle et implémentation en java

Dony, Isabelle

Award date:
2000

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
rue Grandgagnage, 21
B-5000 Namur

Un logiciel d'aide à l'enseignement d'une méthode de programmation : étude conceptuelle et implémentation en java

Isabelle Dony

Mémoire présenté pour l'obtention du grade de licencié en informatique

Promoteur : le Professeur Baudouin Le Charlier

Année académique 1999-2000

Résumé

En première candidature en sciences mathématiques aux Facultés Notre-Dame de la Paix à Namur, nous avons étudié une méthode de programmation qui trouve son inspiration dans les lois de la logique formelle. Le but de cette méthode est d'apprendre à rédiger un programme à partir des spécifications du problème que prétend résoudre ce programme. Cette approche est malheureusement difficile de par son caractère peu intuitif et la rigueur mathématique requise. Un logiciel qui assisterait l'étudiant pas à pas dans sa démarche de construction pourrait rendre ces obstacles pédagogiques plus facilement surmontables. L'objectif de ce mémoire est de réfléchir à la conception de ce logiciel et de le programmer en Java.

Abstract

In my first year of mathematics in FUNDP, we have learned a programming method based on laws of formal logic. The purpose of that method was to learn how to write a program from the specifications of the problem for which the program was aimed at. Unfortunately, as that approach is not intuitive and requires much mathematical rigor, it is difficult to tackle. A software assisting the student step by step in the process of constructing his program, could help him to make his way through. The purpose of this thesis is to approch the problem and implement it in java.

Je tiens à remercier les nombreuses personnes qui m'ont encouragée et aidée tout au long de ce travail.

Tout d'abord, je m'acquitte volontiers d'un devoir de gratitude envers le professeur Baudouin Le Charlier, promoteur de ce mémoire de fin d'études. Ses remarques pertinentes m'ont été d'une aide précieuse.

Un grand merci aux informaticiens qui m'ont aidée à utiliser certains logiciels et à ceux qui ont relu mon travail.

L'aboutissement de ce mémoire doit également beaucoup à ma famille et mes amis qui m'ont soutenue et encouragée durant ces longs mois de labeur. Un remerciement tout particulier à Valéry qui m'a supportée et réconfortée durant les moments difficiles.

Table des matières

Introduction : un défi pédagogique

La programmation est longtemps restée une activité dont l'aspect théorique était mis à l'écart. Son apprentissage se résumait à l'étude d'un langage. Dès lors, la programmation était un processus empirique, artisanal : une fois les quelques mots du langage connus, le reste n'était qu'une affaire de bon sens.

C'est en 1976 , grâce à E.Dijkstra, qu'apparaît une discipline de programmation, avec une méthode qui trouve son inspiration dans les lois de la logique formelle. Le but de cette méthode est d'apprendre à rédiger un programme à partir des spécifications du problème que prétend résoudre ce programme. En un mot, *spécifier* puis *construire* le programme en *démontrant* qu'il résout correctement le problème.

Cette approche est innovatrice, de par l'accent mis sur la rigueur, le raisonnement, la découpe en sous-problèmes, des spécifications formelles, etc. Elle est malheureusement difficile étant donné son caractère peu intuitif et la rigueur mathématique qu'elle nécessite. L'existence d'un logiciel qui suggère pas à pas cette méthode et qui évalue la cohérence et la complétude du raisonnement de l'étudiant pourrait rendre ces obstacles pédagogiques plus facilement surmontables.

L'objectif de ce mémoire est de réfléchir à une conception de ce logiciel, et d'en programmer, en java, une première partie exposant déjà quelques caractéristiques méthodologiques intéressantes. Dans ce rapport, j'expliquerai d'abord la méthode de programmation telle qu'elle est enseignée. Ensuite, je présenterai le logiciel que j'ai réalisé et en détaillerai la conception. Je concluerai enfin par des objectifs qu'il serait essentiel d'atteindre ultérieurement.

Chapitre 1

Une méthode de programmation

Dans le cours de programmation de première candidature en sciences mathématiques aux Facultés Notre-Dame de la Paix, l'accent est mis sur l'apprentissage de l'algorithmique. Ensuite seulement, est enseignée la syntaxe d'un langage impératif simplifié (le Pascal sans récursivité) permettant de transformer un algorithme en un programme.

C'est dans cette perspective qu'il s'agit de définir la notion restreinte d'algorithme : « un algorithme est un ensemble ordonné de règles précises permettant de construire le résultat quand les données sont correctement choisies » J.Arsac[1]. Ces règles sont organisées à l'aide de structures de contrôle (la séquence, l'action conditionnelle, la répétition).

1.1 Raisonner en termes d'actions et de situations

Jacques Arsac[1] explique le fonctionnement d'un algorithme tel qu'il est défini ci-dessus. Dans cette définition, chaque règle consiste en une action à accomplir. Une *action* a pour effet de modifier une *situation* : à une situation est associée une action, conduisant à une nouvelle situation. C'est ainsi que l'algorithme permet, par modifications successives, de passer de la situation initiale à la situation finale. C'est le principe même de la machine à états finis.

Ces considérations permettent d'élaborer une approche constructive des algorithmes. En effet, analyser un algorithme, c'est d'abord définir avec soin les *situations initiale et finale* (que sait-on, que cherche-t-on ?). S'il n'y a pas de passage évident de l'une à l'autre, on cherche s'il y a une étape intermédiaire qui permettrait d'arriver à la situation finale, et dont on pense qu'elle est plus facile à atteindre que la situation finale. Ou bien, on détermine une situation intermédiaire que l'on sait atteindre à

partir des données, et dont on pense qu'elle nous rapproche du but. S'il apparaît qu'on ne peut pas atteindre le but en une seule fois, alors on cherche une *situation générale.*

C'est en fonction de cette situation générale et des situations initiale et finale qu'on va pouvoir construire les suites d'instructions. Une spécification bien rédigée apporte donc tous les renseignements dont on a besoin pour la construction de l'algorithme. En un mot : *spécifier pour construire.*

Cette manière de structurer un programme en termes de situation assure la validité du celui-ci : on associe à chaque instruction du programme une *assertion* (énoncé d'une *situation* vraie quand l'instruction va être exécutée) et l'*instruction* est la règle qui une fois exécutée, transforme l'assertion qui lui était associée. Il est alors possible de raisonner sur le nombre limité d'opérations qui permettent de passer d'une situation à une autre.

1.2 Cadre d'application

Le public visé est un public composé d'étudiants novices en programmation et d'étudiants ayant suivi des cours d'informatique dans le secondaire.

La classe de problèmes envisagés pour illustrer et appliquer la méthode regroupe uniquement des problèmes très simples. En conséquence, les difficultés rencontrées par les étudiants sont liées au raisonnement et non à la compréhension du problème lui-même. De plus, la résolution de problèmes complexes exigerait de la part des étudiants d'avoir compris et assimilé la méthode de construction proposée, ce qui est justement l'objectif final recherché.

Les seules structures de données utilisées sont :
- les variables
- les tableaux à une dimension
- les constantes

Les concepts algorithmiques utilisés sont :
- l'action de base : l'affectation
- les combinaisons d'actions : la séquence, l'action conditionnelle et la répétition d'une action
- le langage des organigrammes et un langage simplifié du Pascal pour exprimer les concepts ci-dessus.

Les problèmes d'entrées/sorties ne sont pas pris en considération. En fait, les solutions de problèmes constitueront des morceaux de programmes et non des programmes complets. La traduction de l'algorithme doit respecter la syntaxe du Pascal.

1.3 Démarche

La méthode telle qu'elle est enseignée par le Professeur Baudouin Le Charlier en première candidature est basée sur l'étude de la transformation des assertions associées aux données manipulées d'un programme. Entre chaque action l'état des données traitées est modifié.

De manière générale, on écrira :

$$\{P\}\ instruction\ \{Q\}$$

pour dire qu'il suffit qu'on ait la situation P pour que, après exécution de l'instruction, on ait la garantie que la situation Q soit satisfaite.

Cette méthode rigoureuse qui demande une bonne maîtrise de la formulation d'assertions mathématiques, consiste en la succession des étapes décrites ci-dessous.

Spécifications

La spécification se base sur un énoncé proposé qui doit être lui-même suffisamment précis. Elle comporte les parties suivantes :

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
2. La *précondition* du programme (notée *Pré*) ou *situation initiale* qui est l'ensemble des conditions que doivent respecter les 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).
4. L'*invariant* du programme (noté I_A) ou *situation générale*, se construit uniquement pour les programmes ayant une forme itérative et est la description de l'ensemble des conditions vérifiées par les objets à chaque passage dans la boucle (*i.e :* à chaque itération).

Choix des instructions

Sur base de la situation générale, plusieurs suites d'instructions doivent être décrites. Ces suites d'instructions sont

1. L'*initialisation* (notée *Init*) qui est l'ensemble des instructions nécessaires à la vérification de la condition :
   { *Pré* } Init { I_A }

2. La *condition de terminaison* (notée B) et la *clôture* (notée *Clôt*) qui est l'ensemble des instructions nécessaires à la vérification de la condition :
   { I_A et B } Clôt { *Post* }

3. L'*itération* (notée *Iter*) qui est l'ensemble des instructions nécessaires à la vérification de la condition :
   { I_A et $non(B)$ } Iter { I_A }

Insistons sur le fait que la méthodologie se base bien sur la construction par invariant, c'est-à-dire que la construction de suites d'instructions se fait en fonction de la description de la situation générale.

L'organigramme ci-dessous éclaire sur l'enchaînement des situations et instructions.

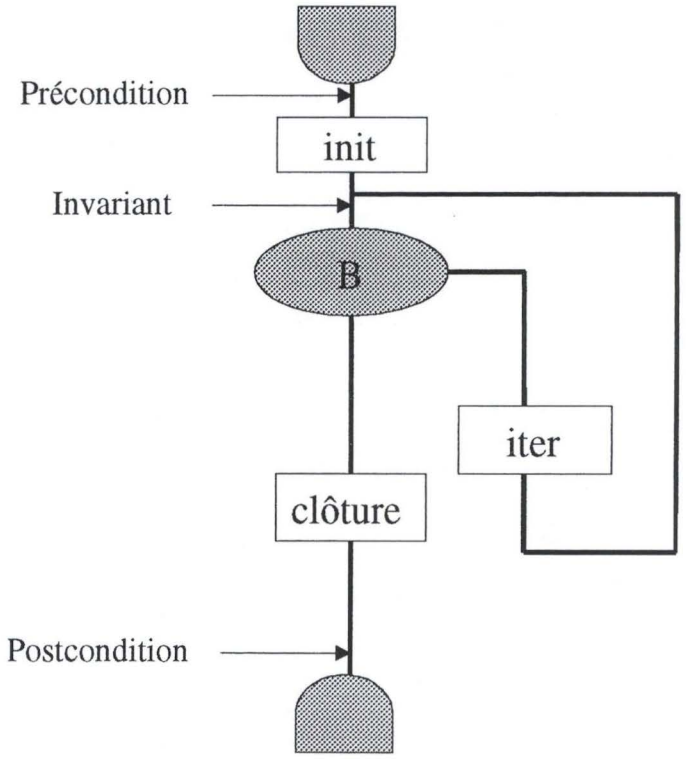

FIG. 1.1 – Le langage de l'organigramme

Vérification de la terminaison

Pour vérifier la validité d'un algorithme, il faut, non seulement, s'assurer que les situations et instructions s'enchaînent correctement, mais il faut aussi prouver que l'algorithme se termine. Pour cela, il suffit de définir une fonction entière et bornée des valeurs des variables, qui croît strictement à chaque exécution de la suite d'instructions *Iter*.

Ecriture du programme

C'est seulement après avoir réalisé toutes ces étapes que l'on peut transformer l'algorithme en Pascal : toutes les suites d'instructions doivent être assemblées de manière à former un seul programme (ou morceau de programme) en respectant la syntaxe du langage.

1.4 Constatations pédagogiques

La méthode de programmation décrite ci-dessus, est enseignée aux facultés Notre-Dame de la Paix de Namur et je l'ai moi-même étudiée pendant ma première candidature en sciences mathématiques appliquées. 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 dans la globalité du problème.

Malheureusement, l'étudiant ne perçoit pas aisément cet avantage considérable, il ne voit qu'une contrainte et ne comprend pas l'intérêt de l'invariant dans l'approche constructive. En effet,

- le raisonnement en termes de situations n'est pas intuitif. L'étudiant préfère naturellement simuler un exemple et construire directement le programme.
  Définir l'invariant est généralement une des plus grandes difficulté. Même si les algorithmes sont développés au cours en donnant à l'invariant un rôle majeur (c'est l'invariant qui permet le développement du corps de la boucle, et pas l'inverse), nombreux sont les étudiants qui ne veulent attribuer à l'invariant qu'un fonction a posteriori.
- la rigueur nécessaire à la formalisation mathématique des assertions est souvent un frein pour le novice de l'université. Même s'il a compris exactement ce qu'il doit mettre dans l'invariant, il a beaucoup de peine à traduire son idée sous forme mathématique.

On constate que cette méthode d'apprentissage de la programmation entre bien dans une formation générale d'universitaire. En effet, cette méthode incite à un comportement rigoureux, enseigne la formalisation mathématique et amène à une bonne capacité d'abstraction. Cependant, cette méthode induit certaines difficultés d'enseignement. Notre objectif est d'utiliser un ordinateur pour surmonter ces obstacles. Idéalement, l'*ordinateur* devrait permettre de montrer aux étudiants résolvant des problèmes de programmation, les erreurs et incohérences qu'ils ne pourraient pas percevoir directement.

Chapitre 2

Aperçu du logiciel

2.1 Objectifs

Le but de ce logiciel est d'illustrer et supporter la méthode d'enseignement de la programmation exposée dans le chapitre précédent et de tenter de la rendre plus accessible. Ce système devrait permettre à l'étudiant de dominer une telle méthode de construction de programme par la mise en évidence des éventuelles erreurs de raisonnement ; il doit donc être capable d'assister l'étudiant à tous les stades de l'élaboration de son programme.

Le logiciel que j'ai réalisé n'est qu'un premier pas vers le logiciel souhaité. Dans ce chapitre, je parlerai des fonctionnalités de mon programme tout en mentionnant celles qui n'ont pas été réalisées, mais qui seraient construites de façon similaire (même corps de programme).

2.2 Fonctionnalités

2.2.1 Construction de la spécification

Utilisation d'assertions

- La syntaxe des assertions devra correspondre au maximum à celle étudiée au cours. Le langage devra donc disposer des quantificateurs et d'une large gamme d'opérateurs.
  Exemples : $\forall$, $\exists$, $\cap$, Π , Σ , etc.
  Les primitives que j'ai réalisées sont celles du $\forall$ et du $\exists$, la programmation des

autres devrait se faire de manière très semblable.
- Ensuite, il y a toutes les primitives suffisamment puissantes pour que l'expression d'une situation ne constitue pas un effort important de programmation.
  Exemples : *inchangé*, *initialisé*, *trié par ordre croissant*, *pair*, etc.
  J'ai réalisé les primitives *inchangé* et *initialisé* ; le sytème est conçu de manière à ce que l'ajout d'un nouveau mot dans la syntaxe et d'une nouvelle sémantique ne pose pas de problème.

Fonctions de vérification

Il est utopique d'espérer une vérification de la correction de spécifications par rapport à un énoncé intuitif. On ne peut faire des vérifications qu'à partir d'expressions formelles.

Les vérifications réalisées sont les suivantes :

- les vérifications syntaxiques : le système impose une certaine syntaxe, celle-ci correspond au maximum à celle apprise au cours de programmation. Il faudra la respecter ; en cas d'erreur, le système doit nous en avertir (*cohérence*).
- les objets utilisés sont identifiés par leur nom ; ainsi, il n'existe pas deux objets de même nom(*cohérence*).
- tous les objets utilisés dans la précondition et postcondition doivent avoir été déclarés comme objets principaux (*cohérence*).
- tous les objets déclarés comme objets principaux devront apparaître dans la précondition ou postcondition (*complétude*).
- la précondition ne contient pas d'objets qui ne sont plus référencés dans la postcondition (*complétude*).

2.2.2 Construction de l'algorithme

Utilisation d'un langage Pascal simplifié

La syntaxe se limite à :

- l'affectation, qui se traduit en Pascal par : ":=".
- la séquence d'actions qui a la forme syntaxique suivante :
  "**begin** $A_1, A_2, \ldots$ **end** ".
- et l'action conditionnelle qui s'écrit de la façon suivante :
  if B **then** A_1 **else** A_2 où B est une expression booléenne.

La répétition d'une action ne se traduit pas dans l'algorithme. La principe de la boucle est automatiquement résolu par le système ; l'étudiant "n'a qu'à" rédiger la suite d'instructions de la boucle et la condition de terminaison.

Les entrées/sorties ne sont pas impliquées dans l'apprentissage de la construction d'algorithme. Cependant, ces fonctions sont implicites ; lorsque le système lit par exemple, "a initialisé" dans la précondition, un tableau apparaît avant l'exécution pour permettre l'initialisation de *a*. C'est, en quelque sorte, la substitution du rôle du "READ". De même, l'affichage des résultats remplace le "WRITE".

Fonctions de vérifications

Nous nous limiterons aux vérifications "internes" des instructions, sans tenir compte de la cohérence qui concerne l'enchaînement des spécifications et instructions, celle-ci étant vérifiée pendant l'exécution. Les vérifications concernant les instructions sont les suivantes :

- Les vérifications syntaxiques (*cohérence*).
- les objets utilisés dans les instructions doivent être déclarés (*cohérence*).
- l'expression de droite d'une affectation doit être définie (*i.e :* toutes les variables qu'elle contient doivent être initialisées) (*cohérence*).
- Les types de l'expression de gauche et de l'expression de droite de l'instruction doivent correspondre (*cohérence*).
- Tous les objets déclarés doivent être utilisés (*complétude*).
- L'instruction de clôture doit être utile par rapport à la postcondition ; elle doit affecter des variables qui se trouvent dans la postcondition (*cohérence*).

2.2.3 Exécution de l'algorithme

Initialisation suivant la précondition.

C'est à ce stade qu'on est amené à faire les initialisations exigées par la précondition afin de pouvoir exécuter le programme.

Fonctions de vérifications.

Le logiciel vérifie, transition après transition, la cohérence entre les suites d'instructions et les spécifications :

- La transition {*Precondition* } Init { *Invariant* } :
  Cette première transition a pour but de :
  1. Vérifier si la précondition est bien respectée.
  2. Exécuter la suite d'instructions de *Init.*
  3. Vérifier si l'invariant est satisfait après cette exécution.
- La transition { *Invariant et non*(B) } Iter { *Invariant* } :
  Cette transition va être répétée tant que la condition B ne sera pas satisfaite, l'invariant restant par définition, toujours vérifié. Son rôle est le suivant :
  1. Exécuter la suite d'instructions de *Iter.*
  2. Vérifier si l'invariant est toujours satisfait après cette itération.
- La transition { *Invariant et* B } Clôt { *Postcondition* } :
  Cette transition est atteinte dès que B est satisfait,
  son rôle est le suivant :
  1. Exécuter la suite d'instructions de *Clôt.*
  2. Vérifier si la postcondition est bien satisfaite après cette clôture.

Affichage des résultats

A chaque transition réussie, le logiciel affiche les valeurs de tous les objets résultant de cette étape.

Dans le cas contraire, le logiciel affiche un message d'erreur indiquant l'expression cible de la situation (contenant souvent plus d'une expression) qui n'est pas satisfaite, de telle sorte que l'utilisateur situe exactement le problème.

2.3 Choix du scénario

Comme on l'a dit dans les objectifs, le système doit, dans la mesure du possible, pouvoir contrôler chaque pas de l'étudiant dans la construction de son programme afin de le réorienter en cas d'erreur.

Concrètement, les vérifications de cohérences internes sont appelées chaque fois que l'utilisateur effectue un enregistrement, qu'il s'agisse d'un objet, d'une assertion, d'une condition ou d'une instruction .

Si la vérification se termine avec succès, l'enregistrement a lieu, et le système reste actif. Par contre, si la vérification observe un problème, le système affiche

un message qui cible exactement l'erreur. Le logiciel permet ainsi à l'étudiant de revenir en arrière pour faire les modifications nécessaires selon le message d'erreur. Il fait ainsi preuve de beaucoup de souplesse tout en étant assez strict pour ne pas l'autoriser à poursuivre dans la mauvaise voie.

Les vérifications de complétude sont réalisées lorsque l'étudiant estime qu'il peut exécuter son algorithme. En cas de vérifications positives, l'exécution peut commencer, sinon apparaît un message permettant de savoir où se situe exactement l'erreur de complétude.

Quant aux vérifications de cohérence entre les spécifications et les instructions, elles se font pendant l'exécution, transition après transition (on peut parler de vérification dynamique). Si une suite d'instructions est incompatible avec une situation, l'exécution s'arrête. Les valeurs des variables modifiées sont ajoutées dans le tableau et la partie spécifique de la situation qui n'est pas vérifiée est indiquée dans un message d'erreur.

Les exemples concrets de scénarios sont détaillés dans le chapitre suivant.

2.4 Caractéristiques de l'interface

L'interface est composée de trois panels.On peut jongler facilement avec les panels *Spécifications* et *Instructions*; le fait de séparer ainsi les spécifications des instructions souligne déjà leur différence sémantique(situation/action). Quant au troisième panel, celui de l'exécution, il n'est accessible que quand les vérifications de cohérence et de complétude sont satisfaites.

Les panels sont analysés plus précisément ci-dessous.

2.4.1 Spécifications

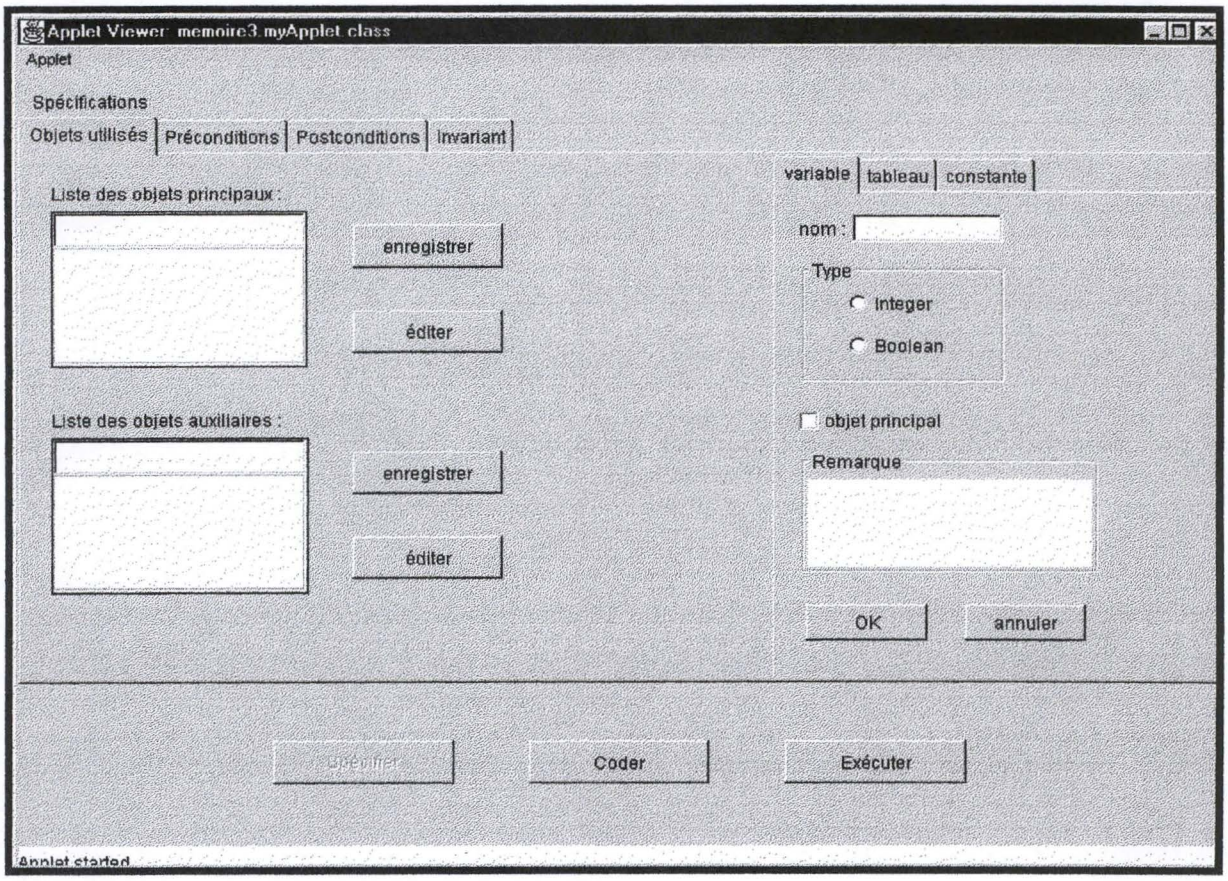

FIG. 2.1 – Le panel des spécifications

- Les onglets permettent d'accéder sans contrainte aux panels des objets utilisés et des situations initiale, finale et générale.

- Dans le panel des objets utilisés, il existe une aide qui permet de déclarer les objets en remplissant des champs de texte sans connaître la syntaxe nécessaire.
- Dans les panels des situations, on pourrait également fournir une aide mais celle-ci n'a pas encore été implémentée.

2.4.2 Instructions

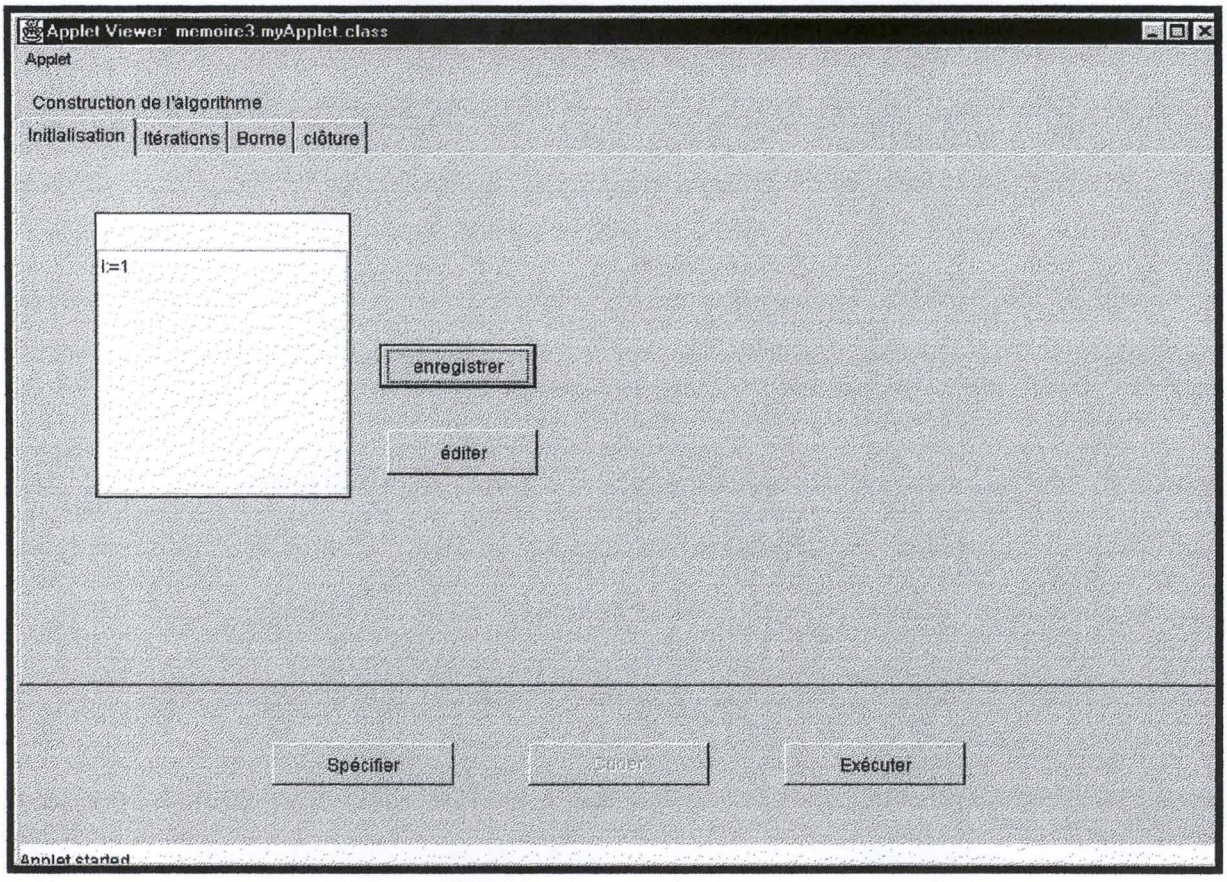

FIG. 2.2 – Le panel des instructions

- Ici encore, les onglets permettent de jongler entre les différentes suites d'instructions.
- On pourrait également fournir une aide qui énumèrerait les variables déclarées pour l'initialisation, les expressions de droite utilisables, etc.
- Rappelons que l'utilisateur peut revenir aux spécifications si par exemple, il avait oublié par mégarde la déclaration d'une variable.

2.4.3 Exécution

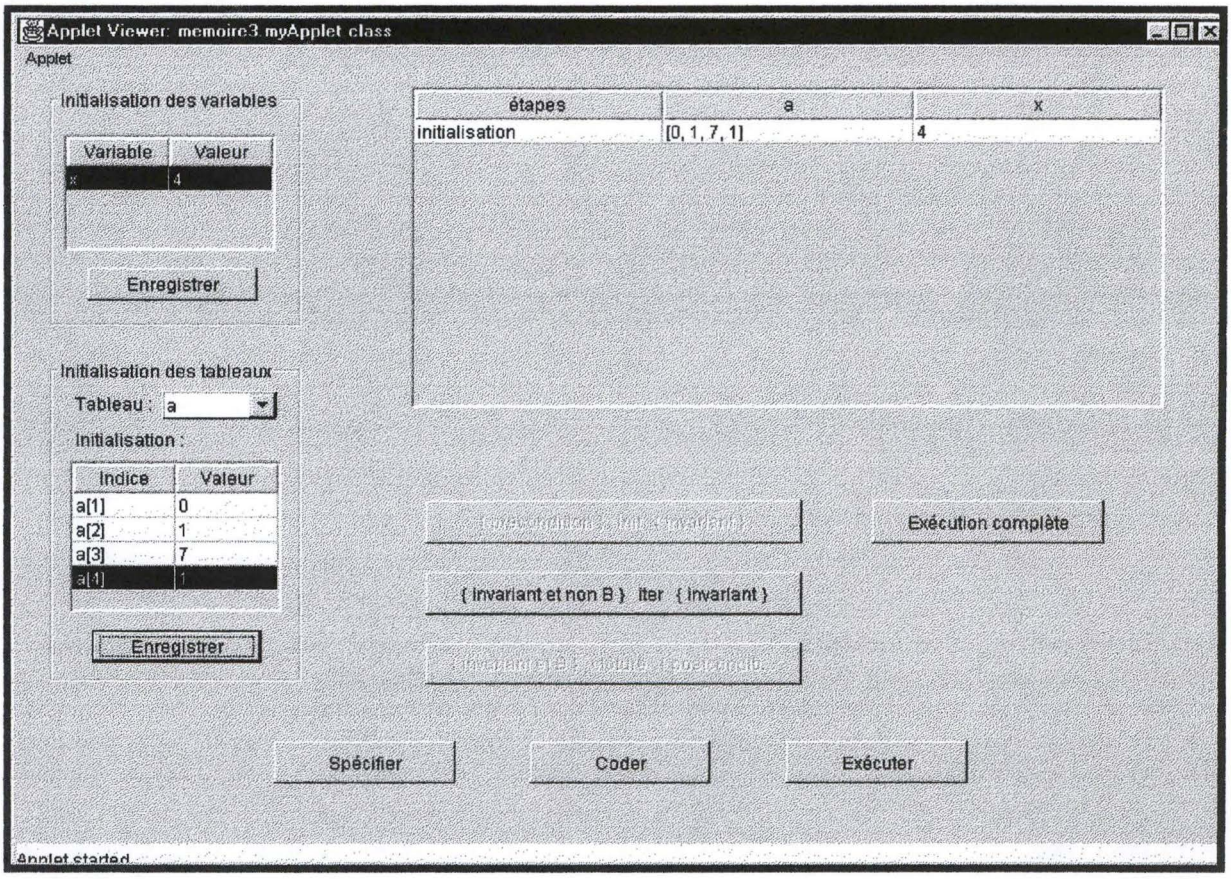

FIG. 2.3 – Le panel des instructions

- Ce panel n'apparaît qu'après avoir été soumis aux vérifications.
- Les tableaux d'initialisation pour variables ou tableaux permettent l'initialisation des objets demandée par la précondition.
- le tableau d'exécution affiche les valeurs des variables à chaque étape de l'algorithme.
- Chaque fois qu'on appelle le panel *Exécution*, celui-ci est réinitialisé.

2.5 Critiques

Pour que le système soit réellement efficace, l'interface devrait permettre toutes les fonctionnalités souhaitées par l'utilisateur. Il faudrait donc que toutes les primitives nécessaires existent. Nous n'en sommes pas là mais le système est conçu de façon à pouvoir les ajouter facilement.

De plus, l'interface doit être compatible avec le profil cognitif de l'utilisateur. Dans notre cas, l'utilisateur est un étudiant qui suit le cours présenté dans le chapitre 2. Le logiciel doit donc être adapté le plus précisément possible à cette méthode de programmation. Le problème qui persiste est celui de la syntaxe : D'une part, dans le logiciel, le $\forall$ se note "pour tout", le $\leq$ se note "<=", etc, le clavier ne contenant pas beaucoup de symboles mathématiques. D'autre part, l'ordinateur exige une syntaxe beaucoup plus rigide qu'un enseignant, ce qui limite les possibilités d'expression.

Chapitre 3

Présentation du logiciel

3.1 Syntaxe concrète

La syntaxe décrite ci-dessous n'est pas complète, elle se limite à celle dont la sémantique est implémentée. En effet, comme dit précédemment, de nombreuses primitives n'ont pas été programmées.

Objets utilisés : déclarations des variables

< *declvar* >::= **var** < *id* > **:** < *type* >
< *decltab* >::= **tab** < *id* > **[** < *bi* > **..** < *bs* > **] : integer**
< *declconst* >::= **const** < *id* > = < *entier* >

< *bi* >::=< *entier* >
< *bs* >::=< *entier* >
< *type* >::= **boolean** | **integer**

< *entier* >::= $([0-9])^+$
< *id* >::= [A–Z, a–z]([A–Z, a–z , 0–9])*

Exemples :

- Pour déclarer une variable booléenne b, on écrit : **var** b : **boolean**.
- Un tableau d'entiers a de 5 éléments sera déclaré de la façon suivante : **tab** $a[1..5]$: **integer**.

Il serait intéressant que les bornes des tableaux puissent être des identifiants repré-

sentant des constantes pour pouvoir par exemple, déclarer un tableau tel que : **tab**$[1 \ldots n]$ **: integer**.

Situations : assertions en langage mathématique

$< situation >::= < quantif >$
$\quad | (< quantif >) (< opor > | < opand >) (< quantif >)$
$\quad | < opnot > (< quantif >)$
$\quad | < bfonct >$
$\quad | < id > = < quantif >$
$\quad | < id > => < quantif >$
$\quad | < bexpr >$

$< bfonct >::= < id >$ **initialisé**
$\quad | < id >$ **inchangé**

$< quantif >::=$ **pour tout** $< id > : < domaine > : < situation >$
$\quad |$ **il existe** $< id > : < domaine > : < situation >$

$< domaine >::=< aexpr >< opppetit >< id >< opppetit >< aexpr >$

$< opppetit >::= < | <=$

Pour les $< situation >$ utilisant un quantificateur, le $< id >$ "quantifié" doit être identique au $< id >$ du $< domaine >$.

Exemples :

- **pour tout** $i : 1 <= i <= 5$ **: pour tout** $j : i < j <= 5$ **:** $a[j] <> x$
- $b =>$ (**il existe** $k : 1 <= k <= 5 : a[k] = x$)

Dans une expression avec quantificateur(s), il faut, d'une part, éviter de choisir pour la(les) variable(s) quantifiée(s), des identifiants qui apparaissent dans les bornes du domaine. D'autre part, dans le cas où il y a plusieurs quantificateurs il vaut mieux choisir des variables quantifiées différentes Par exemple, l'expression
pour tout $i : 1 <= i <= 5$ **: pour tout** $i : i < i <= 5$ **:** $a[i] <> x$ est à éviter [1]

[1] à éviter car ambiguë mais sémantiquement, cette expression signifie la même chose que l'exemple 1.

Instructions

$< instruction >$ ::= $<affectation>$
| $< conditionnelle >$

$< affectation >$ ::= $< lexpr >$ **:=** $(< aexpr > | < bexpr >)$

$< conditionnelle >$ ::=**if** $< bexpr >$ **then** $<affectation>$ **else** $<affectation>$

Une restriction dans la syntaxe est qu'on ne peut pas avoir une suite d'instructions dans une conditionnelle ; ce qui ne serait pourtant pas difficile à implémenter.

$< suite\ instr >$: := **begin**$<instruction_1 >$;$<instruction_2 >$;...**end**

Expressions

Expressions de gauche :

$< lexpr >$::= $< id >$
| $< id >$ **[** $< aexpr >$ **]**

Expressions arithmétiques :

$< aexpr >$::= $< aexpr >< opadd >< aterm >$
| $< aterm >$

$< aterm >$::= $< aterm >< opmult >< aunary >$
| $< aunary >$

$< aunary >$::= $<aélément>$
|− $<aélément>$

$<aélément>$:= $< entier >$
| $< lexpr >$
| $< afonct >$
| $(< aexpr >)$

$< afonct >$::= **Fibo** $(< aexpr >)$
$< opadd >$::= + | −
$< opmult >$::= ∗ | div | /

Attention que les $< afonct >$ ne peuvent pas être utilisées dans les instructions, ce qui n'est traduit nulle part dans la syntaxe.

Expressions booléennes :

$< bexpr >::= < bexpr >< opor >< bterm >$
$\quad | < bterm >$

$< bterm >::= < term >< opand >< bunary >$
$\quad | < bunary >$

$< bunary >::= <bélément>$
$\quad | < opnot ><bélément>$

$<bélément>::= < booleen >$
$\quad | < lexpr >$
$\quad | (< bexpr >)$
$\quad | < aexpr >< oprel >< aexpr >$
$\quad | < bexpr >< opegal >< bexpr >$

$< booleen >::=$ **true** | **false**

$< opor >::=$ **or** | **ou**
$< opand >::=$ **and** | **et**
$< opnot ::=$ **not** | **non**

$< oprel >::= < opegal > | < opinegal >$
$< opegal >::= <> | =$
$< opinegal >::= < | <= | > | >=$

3.2 Exemples de scénarios

3.2.1 Calcul du carré d'un nombre entier

Nous commençons par un exemple très simple ne considérant que des variables entières et utilisant des expressions d'assertions simples.

L'énoncé du problème est le suivant : étant donné un nombre entier n, calculez-en son carré en utilisant la relation $(n+1)^2 = n^2 + 2n + 1$.

Résolvons ce problème en utilisant le logiciel.

Les spécifications

Les objets utilisés.

Etant donné le problème, l'étudiant va commencer par introduire les *variables principales*, celles qui sont indiquées telles quelles dans l'énoncé, et dans ce cas une variable pour le résultat :
- var n : integer ;
- var y : integer ;

Ensuite, l'étudiant devra choisir les *variables auxiliaires* dont il a besoin pour construire l'algorithme. Celles-ci ne doivent pas nécessairement être déclarées à ce moment, il peut d'abord considérer les pré/post-conditions :
- var i : integer ;
- var z : integer ;

La précondition.

Il s'agit d'écrire les conditions vérifées par la valeurs initiales des variables principales :
- $n \geq 0$

La postcondition.

Il s'agit de la relation entre la valeur finale (y) et la valeur initiale (n) :
- $y = n^2$
- n inchangé

L'invariant.

C'est ici qu'intervient toute l'intuition de l'étudiant : à partir de la relation $(n+1)^2 = n^2 + 2n + 1$ et étant donné la postcondition, l'étudiant doit sentir la progression intéressante qui doit se passer dans l'algorithme pour atteindre le résultat.
Il peut considérer que $(i+1)^2$ est défini à partir de i^2, et qu'en itérant jusque n, on

obtient n^2 :
- $0 \leq i \leq n$
- $y = i^2$
- $z = 2 * i + 1$
- n inchangé

Les instructions

Maintenant que les spécifications sont complètes, passons au panel suivant, celui des instructions, nous allons écrire les suites d'instructions, **en fonction** des spécifications qui les "entourent".

Init.

Pour satisfait l'invariant, on initialise les variables de la façon suivante :
- $i := 0$;
- $y := 0$;
- $z := 1$;

Iter.

Si i croît de 1 à chaque itération, il faut considérer que :
- on part de $y_1 = i^2$ et on veut $y_2 = (i+1)^2 = i^2 + 2i + 1 = y_1 + z_1$;
- on part de $z_1 = 2 * i + 1$ et on veut $z_2 = 2 * (i+1) + 1 = 2 * i + 3 = z_1 + 2$

d'où
- $i := i + 1$;
- $y := y + z$;
- $z := z + 2$;

Condition d'arrêt.
- $i = n$

Clôt.
pas de clôture.

L'exécution

Si les vérifications de cohérence et de complétude se passent bien, le système permet l'exécution de l'algorithme. Puisque la précondition contient $n \geq 0$ qui suppose *n initialisé* , un tableau "initialisation de variables" apparaît. L'utilisateur entre ainsi le nombre dont il souhaite trouver le carré.

L'exécution peut alors commencer, transition par transition, affichant dans un tableau les valeurs des variables à chaque étape de l'algorithme.

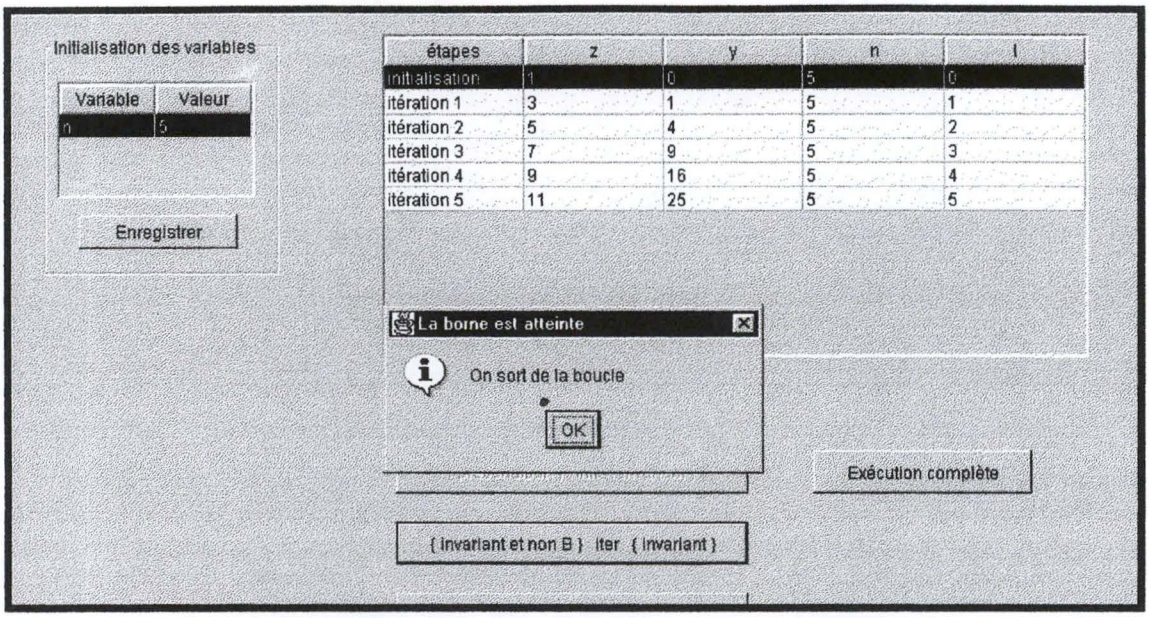

FIG. 3.1 – Exécution correcte de l'algorithme

Utilisation du logiciel quand une erreur se produit

C'est bien sûr dans ce cas que le système démontre toute son utilité, bien que par définition, cette méthode de construction ait été conçue de façon à ne pas générer d'erreur... Le logiciel est utilisé par des débutants !

1. Commençons par un exemple d'erreur de cohérence "interne" aux instructions : Si l'étudiant oublie d'initialiser la variables z et dans l'*Iter*, écrit $y := y + z$, le système va avertir que z n'a pas été initialisé, il suffira alors à l'étudiant de revenir dans *init* afin d'ajouter cette initialisation.(Fig 3.2)
2. Si l'étudiant avait écrit une instruction dans la clôture agissant par exemple, sur la variable i, le système lui aurait dit que c'était inutile, puisqu'on ne mentionne pas cette variable dans la postcondition.
3. Si l'étudiant ne fait pas d'allusion à la variable n dans la postcondition, le système va l'avertir que ce n'est pas normal étant donné qu'elle apparaît dans la précondition.
4. Venons-en aux problèmes de cohérence entre les spécifications et les instructions : Si l'étudiant choisit dans *Iter*, l'instruction $z := z+1$ au lieu de $z := z+2$ la transition $\{ I_A \text{ et } non(B) \}$ Iter $\{ I_A \}$ va mal se passer et le système avertira que précisément, c'est la condition $z = 2 * i + 1$ qui n'est pas vérifiée tout en montrant les résultats de l'itération dans la dernière ligne du tableau.(Fig 3.4)

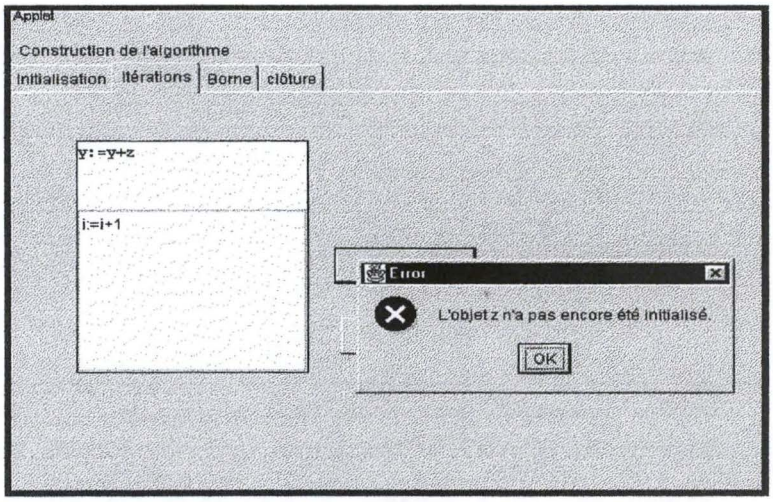

FIG. 3.2 – Erreur de cohérence interne

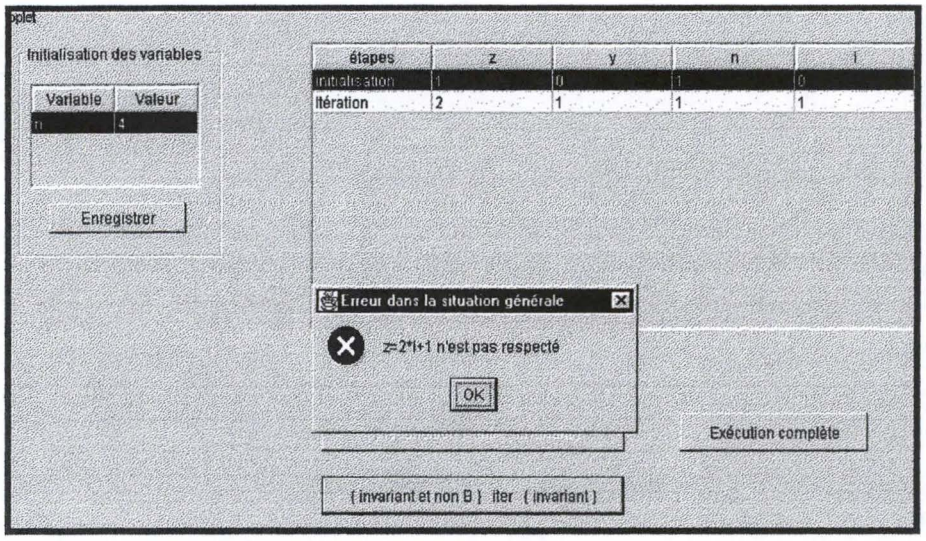

FIG. 3.3 – Erreur de cohérence entre spécifications et instructions

3.2.2 Calcul des nombres de Fibbonaci

Cette exemple utilise la fonction de Fibbonaci,élément qui n'était pas "naturellement" dans la syntaxe du logiciel. Il a donc fallu l'ajouter à la syntaxe permise et lui définir une sémantique.

On demande de calculer le n-ième nombre de Fibonnaci. Ceux-ci sont définis récursivement de la façon suivante :
$F_0 = 0$; $F_1 = 1$; $F_{i+2} = F_{i+1} + F_i \; \forall i \in N$

Les spécifications

Les objets utilisés.

Les variables *principales* représentent le numéro (n) du nombre de Fibonnaci désiré, et la valeur (x) de ce nombre :
- var n : integer ;
- var x : integer ;

Les variables *auxiliaires* sont les suivantes :
- var i : integer ;
- var y : integer ;
- var t : integer ;

La précondition.

- $n \geq 0$

La postcondition.

- n inchangé
- $x = Fibo(n)$

L'invariant.

- $0 \leq i \leq n$
- $Fibo(n) = x * Fibo(i+1) + y * Fibo(i)$
- n inchangé

Les instructions

Init.

- $i := n$;

- $x := 1$;
- $y := 0$;

Iter.

$$Fibo(n) = x * Fibo(i+1) + y * Fibo(i)$$
$$x * (Fibo(i) + Fibo(i-1)) + y * Fibo(i)$$
$$(x+y) * Fibo(i) + x * Fibo(i-1)$$

d'où :
- $i := i - 1$;
- $t := x$;
- $x := x + y$;
- $y := t$;

On voit bien que c'est en **fonction** de l'invariant que les instructions sont construites : on part de l'invariant et on doit le retrouver après les modifications.

Condition d'arrêt.

- $i = 0$

Clôt.
pas de clôture.

L'exécution

- Si l'étudiant souhaite la valeur du nombre de Fibonacci pour un entier négatif, le système avertit qu'il ne satisfait pas à la précondition(Fig 3.4).
- Si l'étudiant choisit l'invariant, pourtant correct, $Fibo(n) = x * Fibo(i) + y * Fibo(i-1)$ avec $1 \leq i \leq n$, l'algorithme n'est pas réalisable dans le cas où $n = 0$. Malheureusement le logiciel ne nous le dit pas...le système ne se rend pas compte de l'incomplétude de l'invariant. C'est une des grandes lacunes du système qui paraît pour le moment insurmontable (en tous cas dans le contexte de travail actuel) : vérifier la complétude de l'invariant.
- Si l'invariant choisi est $Fibo(n) = x * Fibo(i) + y * Fibo(i-1)$ avec $0 \leq i \leq n$ et la condition d'arrêt $i = 0$, l'évaluation de $\{I_A \; et \; B\}$ devient impossible. En effet, celle-ci aurait impliqué le calcul de $Fibo(i-1)$ alors que la fonction de Fibonnaci ne s'applique qu'à des nombres positifs. Le système avertirait l'utilisateur comme le montre la Fig 3.5.

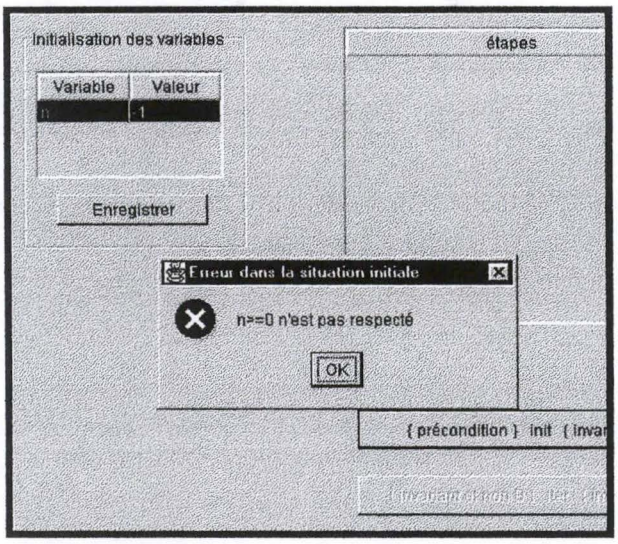

FIG. 3.4 – Erreur de cohérence dans la précondition

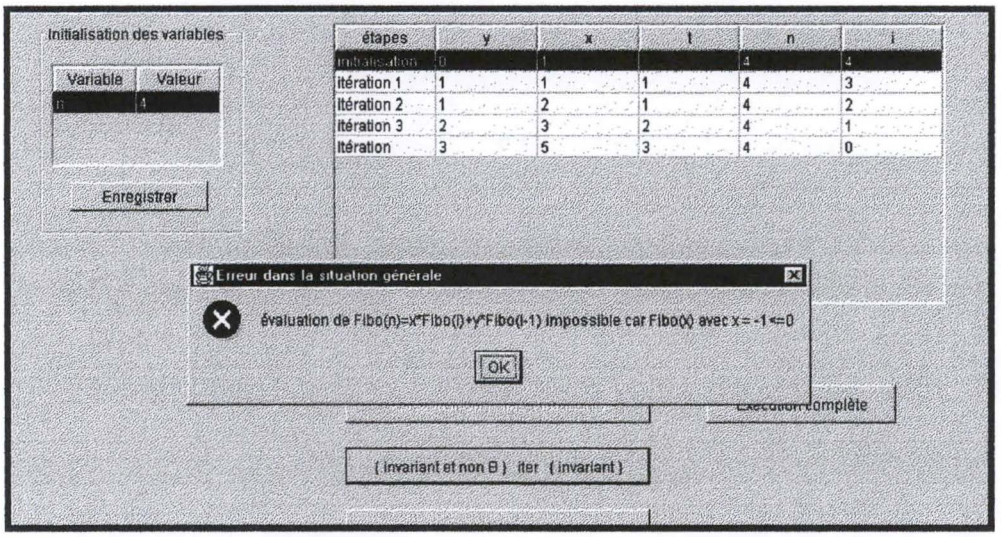

FIG. 3.5 – Erreur de cohérence entre spécifications et instructions

3.2.3 Test de l'appartenance d'un entier à un tableau

Passons à un exemple plus compliqué qui reprend les concepts suivants : le tableau, la variable booléenne et des expressions plus complexes impliquant les quantificateurs $\exists$ et $\forall$.

L'énoncé du problème est le suivant : on donne un tableau de 5 éléments et une valeur entière. On demande de dire si la valeur est présente dans le tableau.

Les spécifications

Les objets utilisés.

- tab a[1..5] : integer ;
- var x : integer ;
- var b : boolean ;

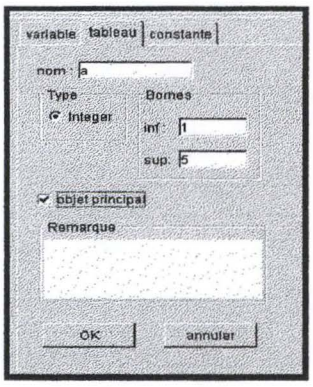

FIG. 3.6 – Initialisation d'un tableau

La précondition.

- a initialisé
- x initialisé

La postcondition.

- $b = (\exists\, i \;:\; 1 \leq i \leq 5 \;:\; a[i] = x)$
- a inchangé
- x inchangé

L'invariant.

- $0 \leq i \leq 5$

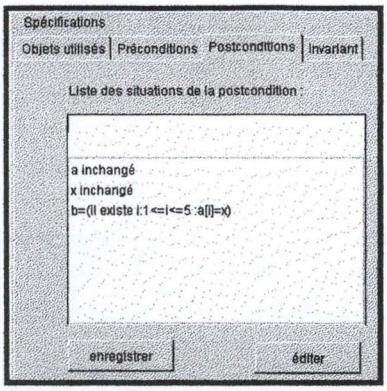

FIG. 3.7 – Introduction de la postcondition

- $\forall j \ : \ i < j \leq 5 \ : \ a[j] \neq x$
- a inchangé
- x inchangé

Les instructions

Init.
- $i := 5$;

Iter.
- $i := i - 1$;

Condition d'arrêt.
- $i = 0$ ou $x = a[i]$

Clôture.
- $b := (i \neq 0)$;

L'exécution

Etant donné la précondition, deux panels apparaissent, l'un servant à l'initialisation de la variable x, l'autre à celle du tableau a.

L'exécution se réalise à nouveau étape par étape, en affichant chaque fois les valeurs du tableau et des variables. Si l'algorithme se déroule normalement, la dernière ligne du tableau contiendra le résultat de l'algorithme, à savoir si le tableau a contient l'élément x ou non, et ce, grâce à la valeur du booléen b.

Utilisation du logiciel quand une erreur se produit

1. Si dans l'invariant, l'utilisateur entre la mauvaise expression suivante : $\forall j$: $i \leq j \leq 5$: $a[j] \neq x$, l'algorithme ne pourra pas fonctionner puisqu'on ne pourra jamais obtenir $\{Invariant\ et\ B\}$.

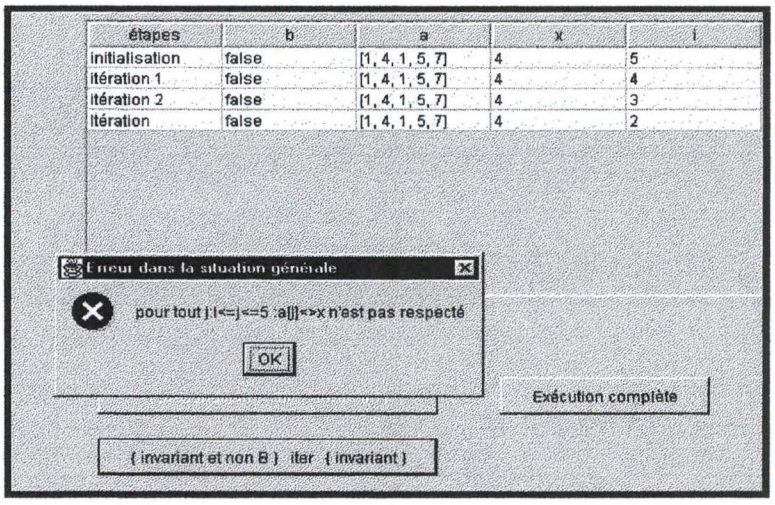

FIG. 3.8 – Erreur dans la construction de l'invariant

2. Si l'étudiant choisit la même borne en inversant l'ordre des conditions, l'évaluation va poser un problème dans le cas où le tableau ne contient pas l'élément x. En effet, le système atteindra la borne $i = 0$ et, évaluant les expressions de gauche à droite, celui-ci va devoir d'abord évaluer l'expression indéterminée, $a[0] = x$.

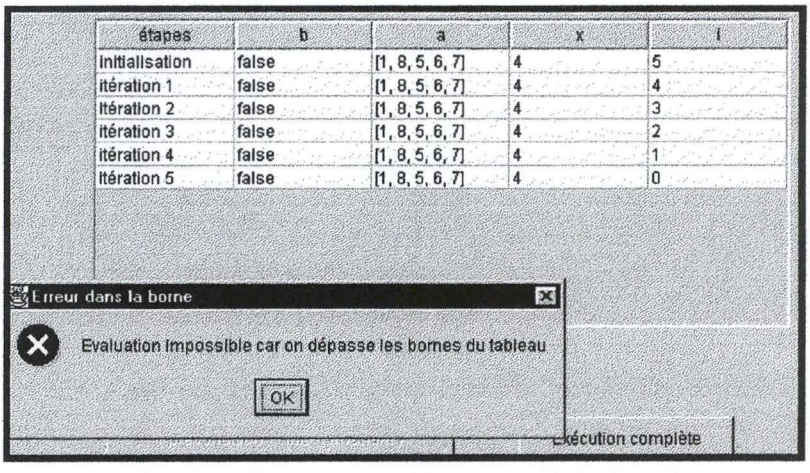

FIG. 3.9 – Erreur dans la construction de la borne

Cet exemple est intéressant de par la mise en évidence du concept d'évaluation d'expressions booléennes de gauche à droite ; cependant on ne respecte pas la sémantique de Pascal puisqu'en Pascal, l'ordre d'évaluation est indéterminé.

3. On sait que pour parcourir entièrement un tableau sans oubli, la *convention des indices* est importante : on peut considérer que
   - l'indice i est avant la borne :

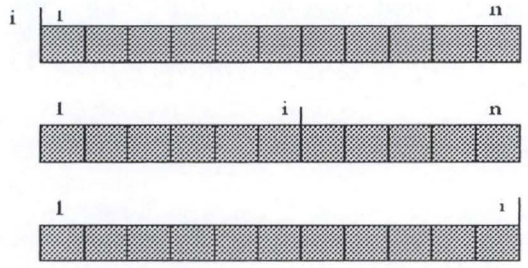

FIG. 3.10 – Parcours du tableau avec l'indice avant la borne

   alors on a $0 \leq i \leq n$;
   - ou que l'indice est après la borne :

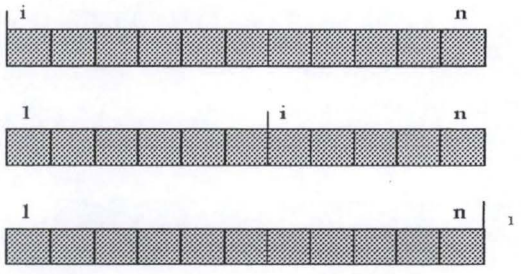

FIG. 3.11 – Parcours du tableau avec l'indice après la borne

   alors on a $1 \leq i \leq n+1$.

Cette notion est importante car ce problème d'indice est source de nombreuses erreurs, et malheureusement le logiciel ne les détecte pas. La lacune de la non-vérification de la complétude de l'invariant revient donc dans ce parcours de tableau.

3.2.4 Recherche dichotomique dans un tableau

Spécifications

- Les pré et postconditions sont les mêmes que dans l'exercice précédent. On a juste une condition en plus dans la précondition : le tableau doit être trié par ordre croissant ; on pourrait créer une primitive *trié par ordre croissant*, mais on peut la traduire facilement sous forme d'assertion mathématique :

$$\forall i : 1 \leq i < 5 : a[i] \leq a[i+1]$$

- L'invariant se comporte tout à fait différemment :
  - $1 \leq g \leq d$
  - $g \leq d \leq 5$
  - $\forall i \ : \ 1 \leq i < g \ : \ a[\ i] < x$
  - $\forall i \ : \ d < i \leq 5 \ : \ a[\ i] > x$
  - $b \Rightarrow (\exists\, i : 1 \leq i \leq 5 : a[i] = x)$
  - a inchangé
  - x inchangé
- Pour les variables auxiliaires, au lieu d'avoir l'indice croissant i (comme dans la recherche séquentielle), on aura besoin de :
  - g indice croissant, partant de la gauche du tableau,
  - d indice décroissant partant de la droite et
  - m indice cherchant le milieu entre g et d.

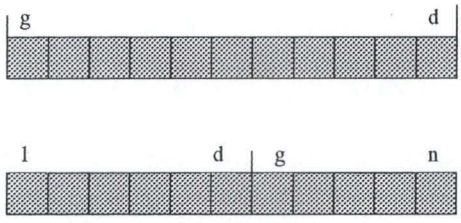

FIG. 3.12 – Parcours des indices : première et dernière étapes

Cet exercice montre encore l'importance de la convention des indices dont malheureusement le logiciel ne perçoit pas les erreurs. Une erreur dans la convention des indices n'est perçue que quand l'exemple d'initialisation choisi la met en évidence. Par exemple, l'algorithme renvoie $b = false$ alors que x est dans le tableau a.

Instructions

Init.

- $g := 1$;
- $d := 5$;
- $b := false$;

Iter.

- $m := (g + d) div\ 2$
- if $(a[m] < x)$ then $g := m + 1$
- if $(a[m] > x)$ then $d := m - 1$
- if $(a[m] = x)$ then $b := true$

Condition d'arrêt.

- $g = d + 1$ ou $b = true$

Clôture.
pas de clôture

Exécution

Dans le cas où l'exécution de l'algorithme se passe bien, le logiciel fonctionne de la façon suivante :

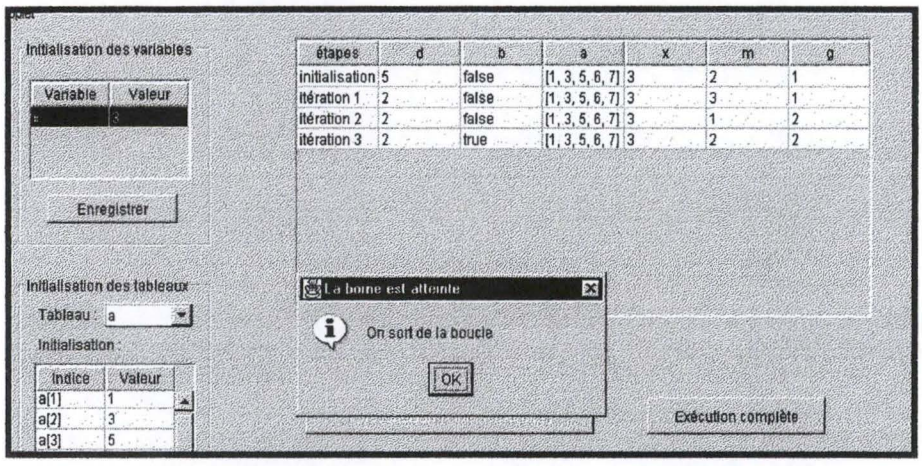

FIG. 3.13 – Exécution correcte de l'algorithme

Le logiciel permet également de mettre en évidence la rapidité d'exécution de cette méthode de recherche par dichotomie par rapport à la méthode de recherche séquentielle décrite plus haut.

Chapitre 4

Analyse conceptuelle du logiciel

4.1 Le logiciel est une application de la méthode de l'invariant

Pour obtenir une grande souplesse dans le logiciel, sa conception est basée sur la notion d'*état cohérent*. L'utilisateur part d'un état cohérent et peut effectuer des modifications puis les enregistrer, en passant librement d'un écran à l'autre . Le sytème permet d'effectuer des modifications et de changer d'écran tant que l'état reste cohérent. Dans le cas contraire, l'utilisateur reste bloqué dans l'écran courant et un message d'erreur est affiché. En fait, le système n'accepte pas d'enregistrer un état incohérent.

Le système est dans un "état cohérent" quand toutes les vérifications de cohérences internes citées dans la section des fonctionnalités sont positives. Attention que les cohérences entre spécifications et instructions ne sont vérifiées que lors de l'exécution finale, on n'en tient donc pas compte quand on parle ici d'état cohérent.

En fait, c'est cet état cohérent qui représente l'*Invariant* du logiciel, et l'*Iter* correspond à une modification de l'utilisateur. (ajout d'une instruction ou d'une assertion, etc.)

L'état cohérent est initialisé par défaut dès qu'on lance le logiciel : la mémoire ne contient pas d'objets utilisés, les spécifications, instructions et condition d'arrêt sont vides. Et donc, le sytème vérifie à ce moment toutes les cohérences. Pendant l'*Init*, l'utilisateur n'a donc rien à faire.

Un progrès de ce logiciel serait d'inclure dans cet *état de cohérence*, les cohérences entre spécifications et instructions.

4.2 Conception détaillée.

Avant toute implémentation de logiciel, une analyse de la conception est indispensable.

Pour ce logiciel en particulier, il faut se fixer la représentation de la mémoire, définir la sémantique des différentes fonctionnalités souhaitées : il s'agit principalement des vérifications de règles de cohérence et de l'exécution de l'algorithme.

Cette analyse peut se faire de multiples façons, j'ai choisi la sémantique dénotationnelle qui permet une traduction assez immédiate lors de l'implémentation en java.

4.2.1 L'environnement et le store

L'**environnement** e d'une description de programme est une fonction des identificateurs vers les *left-values* :

$$e : Id \rightarrow \mathbb{L}_V$$

L'ensemble des *left-values* est défini par :

$$\mathbb{L}_V = \mathbb{L} + (\mathbb{N} \rightarrow \mathbb{L}) + \{undef\}$$

où $\mathbb{L}$ est l'ensemble des locations de mémoires et $\mathbb{N}$ celui des entiers naturels.

Le **store** [1] est une fonction définie comme suit :

$$s : \mathbb{L} \rightarrow \mathbb{R}_V \times \mathbb{T} \times \mathbb{B}$$

où $\mathbb{R}_V = \mathbb{Z} + \mathbb{B}$ définit l'ensemble des *right-values*,
$\mathbb{T} = \{$ "*integer*" , "*boolean*" $\}$ définit l'ensemble des *types* et
$\mathbb{B}$ détermine de façon statique si l'identifiant correspondant est déjà initialisé.

Remarquons que, étant donné le domaine d'application dans lequel on travaille, on pourrait se contenter d'un modèle plus simple sans la notion de *location*. En effet, cette notion est intéressante dans le cas où une location peut correspondre à plusieurs identifiants(appel de procédure ou fonction).

Le modèle défini ci-dessus reste toutefois intéressant dans le cas où, ultérieurement, on ajouterait dans les fonctionnalités du logiciel, les appels de procédure et de fonction.

[1] Le concept de la troisième projection est expliqué plus précisément à la fin du chapitre.

4.2.2 La sémantique dénotationnelle

Cette section introduit la syntaxe abstraite correspondant à la syntaxe concrète définie dans le chapitre 3 et lui attribue une sémantique.

Nous allons commencer par définir la sémantique de chaque expression syntaxique au niveau de l'exécution, sachant que toutes les expressions se trouvent dans un état *cohérent*.

Ensuite, nous définirons la sémantique des vérifications de cohérence de toutes ces expressions. Ces vérifications ne sont pas réalisées lors de l'exécution de l'algorithme mais lors de l'introduction de ces expressions dans le logiciel, ce qui permet d'assurer l'état *cohérent* (invariant du logiciel).

Quelques notations

$$cond(B, A_1, A_2)$$

signifie que si la condition B est satisfaite alors on fait l'action A_1 sinon on fait l'action A_2.

On définit s_1 comme étant la première projection de s :

$$s_1 : \mathbb{L}_V \rightarrow \mathbb{R}_V$$

On définit s_2 comme étant la seconde projection de s :

$$s_2 : \mathbb{L}_V \rightarrow \mathbb{T}$$

On définit s_3 comme étant la troisième projection de s :

$$s_3 : \mathbb{L}_V \rightarrow \mathbb{B}$$

Sémantique d'une expression de gauche

Syntaxe abstraite :

lexpr ::= *id*
| *id*[*aexpr*]

Sémantique :

$$\mathcal{L} : \{lexpr\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{L}v$$

$$\mathcal{L}[\![id]\!] e\ s = e(id) \tag{4.1}$$

$$\mathcal{L}[\![id[aexpr]]\!] e\ s = cond(b, (\mathcal{L}[\![id]\!] e\ s\ )(\mathcal{A}[\![aexpr]\!] e\ s\ )\ ,\ undef\ ) \tag{4.2}$$

où $b = \mathcal{C}heck\mathcal{B}ornes[\![aexpr]\!] f\ e\ s$ tel que $f = \mathcal{L}[\![id]\!] e\ s$ et

$$\mathcal{C}heck\mathcal{B}ornes : \{aexpr\} \rightarrow (\mathbb{N} \rightarrow \mathbb{L}) \rightarrow e \rightarrow s \rightarrow \mathbb{B}$$

$$\mathcal{C}heck\mathcal{B}ornes[\![aexpr]\!] f\ e\ s = \mathcal{A}[\![aexpr]\!] e\ s\ \in dom(f) \tag{4.3}$$

Sémantique d'une expression arithmétique

Syntaxe abstraite :

$$\begin{aligned} aexpr ::= &\ entier \\ &| \ lexpr \\ &| \ -\ aexpr \\ &| \ aexpr\ aop\ aexpr \\ &| \ afonct \end{aligned}$$

Sémantique :

$$\mathcal{A} : \{aexpr\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{Z} + \{error\}$$

$$\mathcal{A}[\![entier]\!] e\ s = entier \tag{4.4}$$

$$\mathcal{A}[\![lexpr]\!] e\ s = s_1(\mathcal{L}[\![lexpr]\!] e\ s) \tag{4.5}$$

$$\begin{aligned} &\mathcal{A}[\![aexpr\ aop\ aexpr]\!] e\ s = (\mathcal{A}[\![aexpr]\!] e\ s)\ \mathcal{O}[\![aop]\!] (\mathcal{A}[\![aexpr]\!] e\ s) \\ &\mathcal{A}[\![-\ aexpr]\!] e\ s = -(\mathcal{A}[\![aexpr]\!] e\ s) \end{aligned} \tag{4.6}$$

$$\mathcal{A}[\![afonct]\!] e\ s = \mathcal{F}[\![afonct]\!] e\ s \tag{4.7}$$

où $\mathcal{F}[\![afonct]\!] e\ s$ correspond à la sémantique de l'appel de fonction $afonct$.[2]

[2]nous n'approfondirons pas le sujet dans ce travail

Sémantique d'une expression booléenne

Syntaxe abstraite :

$$
\begin{aligned}
< bexpr >::= \; & booleen \\
| \; & lexpr \\
| \; & \textbf{not } bexpr \\
| \; & bexpr \; bop \; bexpr \\
| \; & aexpr \; oprel \; aexpr \\
| \; & bexpr \; oprel \; bexpr
\end{aligned}
$$

Sémantique :

$$\mathcal{B} : \{bexpr\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B} + \{error\}$$

$$\mathcal{B}[\![booleen]\!] e \; s = booleen \tag{4.8}$$

$$\mathcal{B}[\![id]\!] e \; s = s_1(e(id)) \tag{4.9}$$

$$
\begin{aligned}
&\mathcal{B}[\![bexpr_1 \textbf{ ou } bexpr_2]\!] e \; s = cond(\neg(\mathcal{B}[\![bexpr_1]\!] e \; s), \mathcal{B}[\![bexpr_2]\!] e \; s, true) \\
&\mathcal{B}[\![bexpr_1 \textbf{ et } bexpr_2]\!] e \; s = cond(\mathcal{B}[\![bexpr_1]\!] e \; s, \mathcal{B}[\![bexpr_2]\!] e \; s, false) \\
&\mathcal{B}[\![\textbf{not } bexpr]\!] e \; s = not \; (\mathcal{B}[\![bexpr]\!] e \; s)
\end{aligned} \tag{4.10}
$$

$$\mathcal{B}[\![aexpr_1 \; oprel \; aexpr_2]\!] e \; s = (\mathcal{A}[\![aexpr_1]\!] e \; s) \; \mathcal{O}[\![oprel]\!] (\mathcal{A}[\![aexpr_2]\!] e \; s) \tag{4.11}$$

$$\mathcal{B}[\![bexpr_1 \; oprel \; bexpr_2]\!] e \; s = (\mathcal{B}[\![bexpr_1]\!] e \; s) \; \mathcal{O}[\![oprel]\!] (\mathcal{B}[\![bexpr_2]\!] e \; s) \tag{4.12}$$

Sémantique d'une instruction

Syntaxe abstraite :

$$
\begin{aligned}
instr \; ::= \; & affect \\
| \; & \textbf{if } bexpr \textbf{ then } affect \textbf{ else } affect \\
| \; & (instr)^*
\end{aligned}
$$

$$affect ::= lexpr{:=}(aexpr|bexpr)$$

Sémantique :

$$\mathcal{I} : \{instr\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{S}$$

$$\mathcal{I} : [\![lexpr := (aexpr|bexpr)]\!] e\ s = s_1(l/rv) \tag{4.13}$$

avec $rv = \mathcal{A}\ [\![aexpr]\!] e\ s$ ou $rv = \mathcal{B}\ [\![bexpr]\!] e\ s$
$l = \mathcal{L}\ [\![lexpr]\!] e\ s$

$$\mathcal{I} : [\![\textbf{if } bexpr \textbf{ then } instr_1 \textbf{else } instr_2]\!] e\ s = cond(b, i_1, i_2) \tag{4.14}$$

avec $b = \mathcal{B}\ [\![bexpr]\!] e\ s$
$i_1 = \mathcal{I}\ [\![instr_1]\!] e\ s$
$i_2 = \mathcal{I}\ [\![instr_2]\!] e\ s$

$$\mathcal{I} : [\![instr_1, instr_2, \ldots, instr_n]\!] e\ s = \mathcal{I}\ [\![instr_2, \ldots, instr_n]\!] e\ s' \tag{4.15}$$

avec $s' = \mathcal{I}\ [\![instr_1]\!] e\ s$

Sémantique d'une assertion

Syntaxe abstraite :

situation ::= | *bexpr*
| *domaine*
| *bfonct*
| *quantif*
| *id* = *quantif*
| *id* => *quantif*
| (*quantif*) *bop* (*quantif*)
| *bnot* (*quantif*)
| (*situation*)*

domaine ::= *aexpr ppt id ppt aexpr*

bfonct ::= |*id* **initialisé**
|*id* **inchangé**

quantif ::= **pour tout** *id* : *aexpr ppt id ppt aexpr* : *situation*
| **il existe** *id* : *aexpr ppt id ppt aexpr* : *situation*

Sémantique :

$$\mathcal{S} : \{situation\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

$$\mathcal{S}[\![bexpr]\!] e\ s = \mathcal{B}[\![bexpr]\!] e\ s \tag{4.16}$$

Domaine, double inéquation :

$$\mathcal{S}[\![aexpr_1\ ppt_1\ id\ ppt_2\ aexpr_2]\!] e\ s = eq_1 \wedge eq_2 \tag{4.17}$$

avec $eq_1 = \mathcal{B}[\![aexpr_1\ ppt_1\ id]\!] e\ s$
$eq_2 = \mathcal{B}[\![id\ ppt_2\ aexpr_2]\!] e\ s$

Les fonctions booléennes

$$\begin{array}{l} \mathcal{S}[\![id(variable)\ \textbf{initialise}]\!] e\ s = s_1(l) \neq noval \\ \mathcal{S}[\![id(tableau)\ \textbf{initialise}]\!] e\ s = \forall l \in codom(f) : s_1(l) \neq noval \end{array} \tag{4.18}$$

$$\begin{array}{l} \mathcal{S}[\![id(variable)\ \textbf{inchange}]\!] e\ s = s_0(l) = s(l) \\ \mathcal{S}[\![id(tableau)\ \textbf{inchange}]\!] e\ s = \forall l \in codom(f) : s_0(l) = s(l) \end{array} \tag{4.19}$$

avec $f = \mathcal{L}[\![id]\!] e\ s$
et s_0 est le store à la précondition

Les assertions contenant un quantificateur :[3]

$$\begin{array}{l} \mathcal{S}[\![\textbf{pour tout}\ id : aexpr_1\ ppt_1\ id\ ppt_2\ aexpr_2 : situation]\!] e\ s = \\ \forall i : b_{inf}\ o_1\ i\ o_2\ b_{sup} : \mathcal{S}[\![situation]\!] e'\ s_i \end{array} \tag{4.20}$$

$$\begin{array}{l} \mathcal{S}[\![\textbf{il existe}\ id : aexpr\ ppt\ id\ ppt\ aexpr : situation]\!] e\ s = \\ \exists i : b_{inf}\ o_1\ i\ o_2\ b_{sup} : \mathcal{S}[\![situation]\!] e'\ s_i \end{array} \tag{4.21}$$

avec $b_{inf} = \mathcal{A}[\![aexpr_1]\!] e\ s$
$b_{sup} = \mathcal{A}[\![aexpr_2]\!] e\ s$
$o_1 = \mathcal{O}[\![ppt_1]\!] e\ s$
$o_2 = \mathcal{O}[\![ppt_2]\!] e\ s$
$e' = \mathcal{D}[\![\textbf{var}\ id : \textbf{integer}\ ]\!] e\ s$
$s_i = s(e'(id)/i)$

$$\begin{array}{l} \mathcal{S}[\![id = quantif]\!] e\ s = (\mathcal{B}[\![id]\!] e\ s = \mathcal{S}[\![quantif]\!] e\ s) \\ \mathcal{S}[\![id => quantif]\!] e\ s = (\neg(\mathcal{B}[\![id]\!] e\ s) \vee \mathcal{S}[\![quantif]\!] e\ s) \end{array} \tag{4.22}$$

[3] $\mathcal{D}[\![\textbf{var}\ id : \textbf{integer}\ ]\!] e\ s$ correspond à la sémantique d'une déclaration, i.e : l'ajout d'un objet utilisé dans l'environnement

$$\begin{aligned}\mathcal{S}[\![\, quantif_1 \textbf{ et } quantif_2 \,]\!]\, e\ s &= (\mathcal{S}[\![\, quantif_1 \,]\!]\, e\ s \wedge \mathcal{S}[\![\, quantif_2 \,]\!]\, e\ s)\\ \mathcal{S}[\![\, quantif_1 \textbf{ ou } quantif_2 \,]\!]\, e\ s &= (\mathcal{S}[\![\, quantif_1 \,]\!]\, e\ s \vee \mathcal{S}[\![\, quantif_2 \,]\!]\, e\ s)\end{aligned} \tag{4.23}$$

Les suites d'assertions : [4]

$$\mathcal{S}[\![\, sit_1, sit_2, \ldots, sit_n \,]\!]\, e\ s = cond(s_1, s_2, false) \tag{4.24}$$

avec $s_1 = \mathcal{S}[\![\, sit_1 \,]\!]\, e\ s$
$s_2 = \mathcal{S}[\![\, sit_2, \ldots, sit_n \,]\!]\, e\ s$

Toute la sémantique définie ci-dessus s'applique quand le système se trouve dans un état *cohérent*(comme expliqué dans la section précédente). Quand l'utilisateur fait une modification de cet état, des méthodes de vérifications sont lancées; leurs sémantiques sont définies ci-dessous.

Commençons par les vérifications concernant les expressions de gauche, elles sont plus complexes.

L'expression de gauche

Sémantique de la vérification d'existence :

$$\mathcal{C}heck\mathcal{E}xist : \{lexpr\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

$$\mathcal{C}heck\mathcal{E}xist[\![\, id \,]\!]\, e\ s = id \in dom(e)$$

$$\begin{aligned}\mathcal{C}heck\mathcal{E}xist[\![\, id[aexpr] \,]\!] = &\mathcal{C}heck\mathcal{E}xist[\![\, id \,]\!]\, e\ s\\ &\wedge \mathcal{C}heck\mathcal{E}xist[\![\, aexpr \,]\!]\, e\ s\\ &\wedge cond(\mathcal{A}[\![\, aexpr \,]\!]\, e\ s \neq noval, \mathcal{C}heck\mathcal{B}ornes[\![aexpr \,]\!]\, f\ e\ s, true)\end{aligned}$$

où $f = \mathcal{L}[\![\, id \,]\!]\, e\ s$ en s'assurant que id correspond bien à un tableau.

Sémantique de la vérification d'initialisation :

$$\mathcal{C}heck\mathcal{I}nit : \{lexpr\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

$$\mathcal{C}heck\mathcal{I}nit[\![\, lexpr \,]\!]\, e\ s = (s_3(\mathcal{L}[\![lexpr \,]\!]\, e\ s) = true)$$

[4] L'évaluation des assertions s'arrête dès que l'une s'avère être fausse.

Les expressions

$expr$ ::=$aexpr$
| $bexpr$

Sémantique des vérifications d'existence et d'initialisation [5]

$$\mathcal{C}heck\mathcal{E}xist : \{expr\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

$$\mathcal{C}heck\mathcal{I}nit : \{expr\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

On ne définit que la sémantique des vérifications de l'existence car les vérifications de l'initialisation ont une construction identique.

$\mathcal{C}heck\mathcal{E}xist \llbracket entier \rrbracket\ e\ s = true$

$\mathcal{C}heck\mathcal{E}xist \llbracket booleen \rrbracket\ e\ s = true$

$\mathcal{C}heck\mathcal{E}xist \llbracket expr\ op\ expr \rrbracket\ e\ s = \mathcal{C}heck\mathcal{E}xist \llbracket expr \rrbracket\ e\ s\ \wedge \mathcal{C}heck\mathcal{E}xist \llbracket expr \rrbracket\ e\ s$

$\mathcal{C}heck\mathcal{E}xist \llbracket op\ expr \rrbracket\ e\ s = \mathcal{C}heck\mathcal{E}xist \llbracket expr \rrbracket\ e\ s$

Les instructions

Sémantique des vérifications d'existence et d'initialisation[4]

$$\mathcal{C}heck\mathcal{E}xist : \{instr\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$
$$\mathcal{C}heck\mathcal{I}nit : \{instr\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

On applique le même argument que précédemment : on définit uniquement la sémantique pour le $\mathcal{C}heck\mathcal{E}xist$.

$$\begin{aligned}\mathcal{C}heck\mathcal{E}xist \llbracket lexpr := rexpr \rrbracket\ e\ s = {} & \mathcal{C}heck\mathcal{E}xist \llbracket lexpr \rrbracket\ e\ s \\ & \wedge \mathcal{C}heck\mathcal{E}xist \llbracket rexpr \rrbracket\ e\ s \\ & \wedge \mathcal{C}heck\mathcal{I}nit \llbracket rexpr \rrbracket\ e\ s\end{aligned}$$

$$\begin{aligned}\mathcal{C}heck\mathcal{E}xist \llbracket \textbf{if } bexpr \textbf{ then } instr_1 \textbf{else } instr_2 \rrbracket\ e\ s = {} & \mathcal{C}heck\mathcal{E}xist \llbracket bexpr \rrbracket\ e\ s \\ & \wedge \mathcal{C}heck\mathcal{I}nit \llbracket bexpr \rrbracket\ e\ s \\ & \wedge \mathcal{C}heck\mathcal{E}xist \llbracket instr_1 \rrbracket\ e\ s \\ & \wedge \mathcal{C}heck\mathcal{E}xist \llbracket instr_2 \rrbracket\ e\ s\end{aligned}$$

[5] Les vérifications de cohérences sont font de gauche à droite et s'arrêtent dès qu'une vérification est fausse ; cette notion n'est pas traduite dans la sémantique.

Sémantique des vérifications des types [6]

$$\mathcal{C}heck\mathcal{T}ype : \{instr\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

$\mathcal{C}heck\mathcal{T}ype [\![lexpr_1 := lexpr_2]\!] e\ s = (s_2(\mathcal{L} [\![lexpr_1]\!] e\ s) = s_2(\mathcal{L} [\![lexpr_2]\!] e\ s))$
$\mathcal{C}heck\mathcal{T}ype [\![lexpr := aexpr]\!] e\ s = (s_2(\mathcal{L} [\![lexpr]\!] e\ s) = "integer")$
$\mathcal{C}heck\mathcal{T}ype [\![lexpr := bexpr]\!] e\ s = (s_2(\mathcal{L} [\![lexpr]\!] e\ s) = "boolean")$

$\mathcal{C}heck\mathcal{T}ype [\![\textbf{if } bexpr \textbf{ then } instr_1 \textbf{else } instr_2]\!] e\ s = \mathcal{C}heck\mathcal{T}ype [\![instr_1]\!] e\ s$
$\wedge\ \mathcal{C}heck\mathcal{T}ype [\![instr_2]\!] e\ s$

Les situations

Sémantique des vérifications d'existence[5]

$$\mathcal{C}heck\mathcal{E}xist : \{situation\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

$\mathcal{C}heck\mathcal{E}xist [\![aexpr_1\ ppt_1\ id\ ppt_2\ aexpr_2]\!] e\ s = \mathcal{C}heck\mathcal{E}xist [\![aexpr_1]\!] e\ s$
$\wedge\ \mathcal{C}heck\mathcal{E}xist [\![id]\!] e\ s$
$\wedge\ \mathcal{C}heck\mathcal{E}xist [\![aexpr_2]\!] e\ s$

$\mathcal{C}heck\mathcal{E}xist [\![id \textbf{ initialisé}]\!] e\ s = id \in dom(e)$

$\mathcal{C}heck\mathcal{E}xist [\![id \textbf{ inchangé}]\!] e\ s = id \in dom(e)$

$\mathcal{C}heck\mathcal{E}xist [\![\textbf{(pour tout|il existe)}\ id : aexpr_1\ ppt_1\ id\ ppt_2\ aexpr_2 : situation]\!] e\ s =$
$\mathcal{C}heck\mathcal{E}xist [\![aexpr_1]\!] e\ s$
$\wedge\ \mathcal{C}heck\mathcal{E}xist [\![aexpr_2]\!] e\ s$
$\wedge\ \mathcal{C}heck\mathcal{E}xist [\![situation]\!] e'\ s$
où $e' = \mathcal{D} [\![\textbf{var } id : \textbf{integer}]\!] e\ s$

$\mathcal{C}heck\mathcal{E}xist [\![id\ (= \mid =>)\ quantif]\!] e\ s = \mathcal{C}heck\mathcal{E}xist [\![id]\!] e\ s$
$\wedge\ \mathcal{C}heck\mathcal{E}xist [\![quantif]\!] e\ s$

$\mathcal{C}heck\mathcal{E}xist [\![quantif_1\ \textbf{(et|ou)}\ quantif_2]\!] e\ s = \mathcal{C}heck\mathcal{E}xist [\![quantif_1]\!] e\ s$
$\wedge\ \mathcal{C}heck\mathcal{E}xist [\![quantif_2]\!] e\ s$

[6] Les vérifications de cohérences sont font de gauche à droite et s'arrêtent dès qu'une vérification est fausse ; cette notion n'est pas traduite dans la sémantique.

Rôle de la troisième projection du store

Afin de savoir vérifier de manière statique qu'une expression est bien définie, c'est-à dire que les *lexpr* qu'elle contient sont bien initialisées, j'ai choisi l'option suivante.

Lors de la déclaration des objets, s_3 renvoie false pour la(les) locations attribuées à l'objet. Dès que une *lexpr* apparaît dans le côté gauche d'une instruction, s_3 est modifiée : $s_3(\mathcal{L}[\![lexpr]\!] e\ s) = true$

Si l'initialisation se fait dans une instruction conditionnelle, le système agit comme Java, *i.e :* si la *lexpr* est la même dans l'affectation du *then* que dans l'affectation du *else*, alors $s_3(\mathcal{L}[\![lexpr]\!] e\ s) = true$, sinon s_3 reste inchangée.

L'inconvénient est que cette manière de procéder risque de repérer des erreurs qui n'existent pas : par exemple, dans le cas où la condition de la conditionnelle serait toujours vraie (ou toujours fausse) et que l'initialisation se ferait uniquement dans l'affectation du *then* (ou resp. du *else*).

Chapitre 5

Implémentation

5.1 Architecture globale

Le coordinateur du logiciel est la classe **Principal** qui contient (et initialise) l'environnement (mémoire). Il déclenche les modules suivants qui sont structurés sur base des écrans : **spécifications**, **instructions** et **exécution**. Ceux-ci se composent chacun d'un sous-module **ihm** et d'un sous-module **vérifications**. Ces sous-modules utilisent d'une part, un module **syntaxe** dont le rôle est de parser les expressions, et d'autre part, une module **objets** qui contient des classes représentant les différents composants de l'algorithme, les spécifications et les instructions (selon la syntaxe abstraite). Des instances de ces classes vont être créées par le parseur, tandis que les méthodes qui y sont développées sont utilisées pour les vérifications de cohérence et de complétude, et pour l'exécution de l'algorithme.

Concrètement, trois outils ont été utilisés :

1. JavaCC dont le rôle est de générer automatiquement un programme parseur en java à partir de la syntaxe concrète définie au chapitre 3.
2. JBuilder qui est logiciel qui génère automatiquement le code Java des interfaces graphiques.
3. Java qui est le langage d'implémentation du logiciel et qui utilisé en particulier pour créer les objets représentant les éléments correspondant à la syntaxe abstraite et pour traduire la sémantique décrite dans le chapitre 4.

Dans la section suivante, nous allons détailler l'implémentation des ces modules et sous-modules, en expliquant quand il se doit, les outils d'implémentation et les notions théoriques nécessaires.

La terminologie sera basée sur Java, le langage d'implémentation du logiciel.

5.2 Architecture détaillée

5.2.1 Représentation de l'environnement et du store

On considère la mémoire comme une table de symboles qui, pour chaque objet (variable, tableau ou constante) renvoie le pointeur vers l'entrée (*Hashtable*). Chaque clé d'accès est un *String*, nom de l'objet utilisé vers lequel elle pointe. Ainsi, l'identifiant de l'objet utilisé correspond directement à sa location de mémoire . Lorsqu'on veut agir sur la valeur d'un objet, on accède à celui-ci via le pointeur renvoyé par la table.

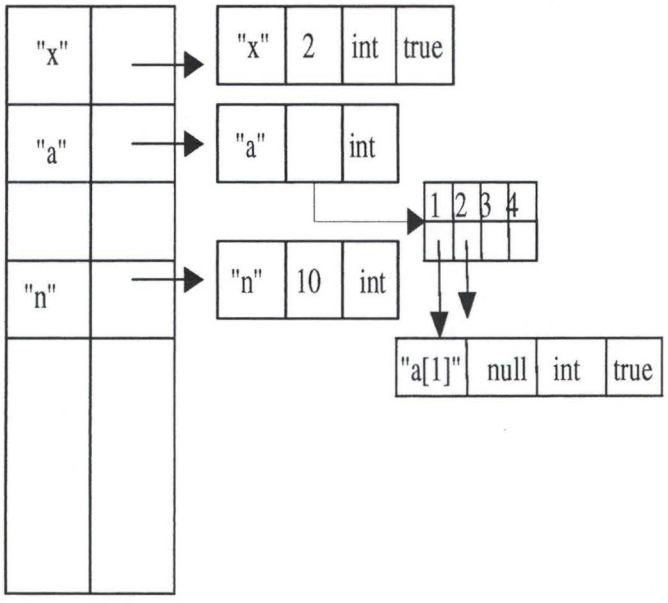

FIG. 5.1 – Représentation de l'environnement et du store

Explicitement, la *fonction de l'environnement* e est représentée par la *Hashtable* même qui "applique" les clés (*identifiants*) aux pointeurs ($\mathbb{L}_V \rightarrow \mathbb{R}_V \times \mathbb{T} \times \mathbb{B}$).

Dans notre représentation, on a $\mathbb{L}_V = \mathbb{L} + (\mathbb{N} \rightarrow \mathbb{L}) + undef$ tels que $\mathbb{L}$ correspond aux origines des pointeurs de la *Hashtable* et $(\mathbb{N} \rightarrow \mathbb{L})$ correspond aux *Hashtables* des tableaux.

La *fonction du store* s est représentée par le pointeur même qui applique l'origine du pointeur (location) à un objet qui contient une valeur, un type et un booléen.

5.2.2 Les module spécifications, instructions et exécution

5.2.2.1 L'interface graphique

Les sous-modules **ihm** créent les interfaces dont les caractéristiques ont été expliquées dans le chapitre 2. L'implémentation a été réalisée en swing en utilisant le logiciel JBuilder3[10]. C'est un environnement de travail qui utilise le langage de programmation java et qui permet la création plus aisée d'applications interactives. L'avantage principal de JBuilder est qu'il évite au programmeur de devoir implémenter les fenêtres et autres objets d'interaction(bouton, listes,....), tout en laissant la possibilité d'intervenir comme on le souhaite dans le code.

5.2.2.2 Les méthodes de vérifications, méthodes appelées par l'interface

Pour les modules **Spécifications** et **Instructions**, les sous-modules de vérifications permettent de contrôler l'invariant (*état cohérent*) après chaque intervention de l'utilisateur sur l'environnement (*i.e :* lors de chaque enregistrement). Pour ce genre de vérifications, on parle d'analyse *statique.*

Quant au sous-module de vérifications du module **Exécution**, il vérifie la cohérence entre spécifications et instructions, de façon *dynamique*, c'est-à-dire qu'il exécute l'algorithme pour faire les vérifications. Ce sous-module contient également des vérifications de complétude qui se font lorsque l'utilisateur veut passer au panel de l'exécution.

L'instance de la mémoire se fait au niveau de la classe *Principal* (au lancement du programme) ; parallèlement, une mémoire ne contenant que les objets déclarés comme principaux est instanciée dans le module **Spécifications**. Cette mémoire que nous appellerons *mémoire des objets principaux* sera utilisée pour toutes les vérifications de complétude.

Les fonctions de vérifications du module spécifications

Le sous-module de vérifications contient une classe nommée *Cohérence* dont une instance est créée en fonction de la mémoire h et la mémoire des objets principaux $h1$. Le but des principales méthodes que contient cette classe est expliqué ci-dessous.

$$Ajouter(String\ o,\ boolean\ b)$$

est utilisée lors de chaque déclaration d'objet et vérifie si l'objet déclaré dans le

String o peut être ajouté à la liste des variables qui se trouvent dans la mémoire h.

Mise à part la vérification syntaxique, les vérifications consistent entre autres à s'assurer qu'il n'existe pas encore de variable du même nom et, dans le cas d'un tableau à contrôler la cohérence des bornes.

Si la vérification est positive, la méthode ajoute l'objet dans la mémoire h et si l'objet est en plus déclaré comme principal, il est aussi ajouté dans la "mémoire des objets principaux" $h1$.

RemoveObjet(String x)

est appelée lors de chaque édition des listes d'objets utilisés ; elle supprime l'objet édité des mémoires h et $h1$.

AjouterSit(String o, Hashtable h)

est utilisée lors de chaque enregistrement d'assertion ; elle vérifie que les objets utilisés dans l'assertion sont bien déclarés. (*CheckExist(h)* défini dans la sémantique des vérifications des situations)

Les fonctions de vérifications du module instructions

Son sous-module de vérifications est également composé d'une classe nommée *Cohérence* qui contient les méthodes de vérifications suivantes :

Ajouter(String o, Vector set, int ent)

est utilisée lors de chaque enregistrement d'instruction ou expression ; elle vérifie si l'instruction ou expression qui se trouve dans le *String o* est bien cohérente.

- Dans le cas d'une instruction,
  il faut s'assurer de l'existence des variables (*CheckExist(h)*),
  vérifier que la partie de droite de l'instruction est bien définie (*CheckInit(h)*)
  et que les types des parties gauche et droite concordent (*CheckType(h)*).
- Dans le cas d'une expression,
  il faut vérifer que tous les éléments sont bien déclarés et initialisés (la vérification des types se fait déjà au niveau de la syntaxe).

RemoveInstr(String o, Vector set, int ent)

est utilisée lors de chaque édition dans les listes d'instructions ou expressions ; elle supprime une instruction éditée (ou expression) tout en vérifiant que les autres instructions resteront cohérentes.

Pour le module exécution

Son sous-module de vérifications contient deux classes que nous expliquons ci-dessous :

Commençons par la classe nommée *Completude* dont l'instance est créée en fonction des spécifications enregistrées et qui contient toutes les méthodes de vérifications de la complétude ; la classe *Principal* appelle ces méthodes avant de passer à l'exécution effective de l'algorithme.

VerifComplSit()

contrôle si les objets utilisés principaux (dans $h2$) sont bien mentionnés dans la précondition ou postcondition.

VerifEntreSit()

s'assure qu'il n'existe pas de variable mentionnée dans la précondition sans être dans la postcondition.

CheckAllInit()

vérifie si touts les objets déclarés sont bien utilisés.

D'autre part, il y a une classe nommée *Cohérence* qui, tout en exécutant l'algorithme, réalise les vérifications entre spécifications et instructions et stoppe l'exécution en cas de non-cohérence.

Les rôles des méthodes principales sont décrites ci-dessous :

$$EvalSit(Vector\ sit)$$

permet de vérifier si la suite d'assertions qui se trouvent dans le vecteur *sit* (en fait, cette suite représentera la situation initiale, générale ou finale) est bien satisfaite étant donné l'environnement dans lequel se trouve le sytème.

La traduction de la relation 4.24 correspondant à la sémantique dénotationnelle d'une suite d'assertions se fait de la manière suivante : le vecteur *sit* contient en fait, des objets *Situation* (voir section suivante) ; on parcourt le vecteur en prenant un à un chaque objet et en lui appliquant sa sémantique. Si la sémantique de l'objet considéré renvoie vrai, on passe au suivant ; si elle renvoie faux, une exception du type *EvalExpr* est initialisée.

$$ExeSuiteInst(Vector\ inst)$$

exécute la suite d'instructions qui se trouvent dans le vecteur *instr* et modifie ainsi l'environnement.

La traduction de la relation 4.15 correspondant à la sémantique dénotationnelle d'une suite d'instructions se fait de la manière suivante : on parcout le vecteur *instr* en prenant tour à tour les objets *Instruction* qu'il contient et en appliquant la sémantique correspondant (*i.e* : exécuter l'instruction). Si une instruction ne peut s'exécuter (ce qui doit être rare étant donné que l'invariant/état cohérent devrait toujours être vérifié), une exception est initialisée ; elle sera du type *ExistException*, *InitException*, ou *TypeException*, ...

$$SI - init - SG()$$

Cette méthode est appelée quand la situation initiale est vérifiée. Elle exécute la suite d'instructions de *Init* (*ExeSuiteInst(Vector init)*), ce qui modifie le store, et vérifie si la situation générale est satisfaite dans ce nouvel environnement (*EvalSit(Vector sit)*).

$$SG - B - clot - SF()$$

Cette méthode est appelée quand la situation générale et la condition d'arrêt B sont vérifiées. Elle exécute la suite d'instructions de *Clôture* (*ExeSuiteInst(Vector clô)*), ce qui modifie le store, et vérifie si la situation finale est satisfaite dans ce nouvel environnement (*EvalSit(Vector sit)*).

$$SG - nonB - iter - SG()$$

Cette méthode est appelée quand la situation générale est vérifiée ; si la condition d'arrêt B est satisfaite, une exception est initialisée pour prévenir qu'on sort de la boucle. Si B n'est pas satisfait, la méthode exécute la suite d'instructions de *Iter* (*ExeSuiteInst(Vector iter)*), ce qui modifie le store, et vérifie si la situation générale est satisfaite dans ce nouvel environnement (*EvalSit(Vector sit)*).

$$SI - SF()$$

Cette méthode est appelée quand la situation initiale est satisfaite. Elle commence par exécuter la suite d'instructions de *Init* et évaluer la situation générale (*SI-init-SG()*). Tant que la borne n'est pas atteinte, elle exécute la suite d'instructions de *Iter* et vérifie que l'invariant est toujours satisfait (*SG-nonB-iter-SG()*). Quand B est satisfait, elle exécute la *clôture* et vérifie la condition finale (*SG-B-clot-SF()*). Quand une situation n'est pas satisfaite, les exceptions sont initialisées comme dans les méthodes expliquées plus haut.

Pour utiliser toutes ces méthodes, il faut créer les objets *Instruction*, *Situation*, *ObU*, ...

Ces objets consistent en deux parties :

- la partie syntaxique qui est le fait de créer ces objets en fonction d'expressions entrées par l'utilisateur (rôle du module **syntaxe** qui est expliqué dans la section suivante).
  → rôle des champs de la classe qui représente l'objet
- la partie sémantique qui est la traduction de la sémantique dénotationnelle détaillée dans le chapitre précédent et qui permet les vérifications et l'exécution de l'algorithme.
  → rôle des méthodes de la classe qui représente l'objet

Dans les sections suivantes, nous allons commencer par détailler le module **syntaxe** qui parse les expressions entrées par l'utilisateur pour en créer des objets. Ensuite seulement, nous expliquerons en quoi consiste le module **objets** syntaxiquement (champs des classes), et sémantiquement (méthodes directement traduites de la sémantiques dénotationnelle).

5.2.3 Le module syntaxe

Le rôle d'un analyseur syntaxique (parseur) est de lire un champ en entrée[1] et de déterminer s'il est conforme à la grammaire fixée.

Dans notre contexte, l'analyseur syntaxique doit retrouver *l'analyse grammaticale* d'un assertion, ou d'une partie d'un programme (déclaration, instruction, expression) et en déceler les erreurs.

Pour créer ce parseur, l'outil utilisé est *Javac Compiler Compiler*. C'est lui qui génère automatiquement le programme parseur en java à partir d'une grammaire spécifiée. En plus d'être un générateur de parseur, JavaCC permet notamment de construire un arbre syntaxique.

Pour construire une grammaire, plusieurs notions sont indispensables : il faut savoir comment une grammaire est définie, comprendre le fonctionnement du parseur utilisé (notion de dérivation), et être capable de modifier la grammaire pour que le parseur retrouve la structure grammaticale de façon déterministe.[7]

5.2.3.1 Notion de grammaire

Une *production* est une paire contenant

- un non terminal $\in V_N$
- une suite de symbole $\in (V_N \cup V_T)^*$

Une *grammaire* est un quadruplet contenant

- l'ensemble de terminaux V_T
- l'ensemble de non terminaux V_N
- l'ensemble fini des productions P
- le symbole de départ $S \in V_N$

Illustration de ces éléments de grammaire :

- V_T = les symboles reconnus lors de l'analyse lexicale :
  Token tels que **if**, les identificateurs (par exemple **x**), **true**, etc.
- V_N = les catégories syntaxiques :
  $< bexpr >$, $< instruction >$, $< equation >$, etc.
- P = les règles de grammaire :
  $< equation >::=< aexpr >< oprel >< aexpr >$,

[1] *inputstream* en terminologie Java.

- S =la catégorie de textes complets :
  $< instruction >$, $< ObU >$, $< bexpr >$, $< situation >$.
  Dans notre cas, le parseur parse plusieurs "grammaires" car on travaille sur chaque élément de l'algorithme séparément.

5.2.3.2 Notion de dérivation

Soient ϕ et ψ deux suites finies d'éléments de $(V_N \cup V_T)$.

On dit que ϕ *produit directement* ψ :

$$\phi \overset{P}{\Rightarrow} \psi$$

ssi
- $\exists (A \to \alpha) \in P$ avec $A \in V_N$
- $\phi = \sigma A \tau$
- $\psi = \sigma \alpha \tau$ avec $\alpha, \tau, \sigma \in (V_N \cup V_T)^*$

Une *dérivation* est une suite

$$S \overset{P_1}{\Rightarrow} \psi_1 \overset{P_2}{\Rightarrow} \psi_2 \Rightarrow \ldots \overset{P_n}{\Rightarrow} \psi_n$$

Terminologie :

- Si $\psi_n \in (V_T)^*$, chaque chaîne qu'on peut obtenir ainsi est une **phrase** de la grammaire G.
- L'ensemble des phrases est le **langage** de G noté $L(G)$.
- Les étapes intermédiaires sont des **proto-phrases**.
- Un non-terminal est **inaccessible** s'il ne fait partie d'aucune proto-phrase.
- Il est **improductif** s'il ne produit aucune chaîne de terminaux

5.2.3.3 Notion d'analyseur syntaxique descendant

Le parseur généré par JavaCC est un analyseur syntaxique descendant (LL).

LL signifie que la dérivation est à *gauche*, c'est-à-dire que c'est une dérivation où, à tout moment, seul le non terminal le plus à gauche est étendu.

Le rôle du parseur sera de retrouver la dérivation à partir d'une expression donnée. Si le parseur est non-déterministe, alors il risque d'y avoir à un moment donné, plusieurs expansions possibles. Laquelle choisir ?

On dit qu'une analyseur syntaxique est $LL(k)$ quand il suffit de *regarder les k premiers symboles en entrée pour pouvoir choisir directement la bonne production.*

Dans la mesure de mes possibilités, étant donné la syntaxe souhaitée, j'ai tenté d'écrire une grammaire $LL(1)$.

Par exemple , la règle de grammaire considérant les objets utilisés est $LL(1)$:

$<$*objet utilisé*$>$::= **var** $<$ *id* $>$:$<$ *type* $>$
| **tab** $<$ *id* $>$ [$<$ *bi* $>$ .. $<$ *bs* $>$] :$<$ *type* $>$
| **const** $<$ *id* $>$=$<$ *entier* $>$

En effet, si *pour chaque non-terminal, chacune des productions commence par un symbole terminal différent, alors la grammaire est LL(1).*

On voit bien qu'il suffit de lire le premier *token* pour savoir de quel type de déclaration on parle.

Toute grammaire LL doit doit satisfaire certaines propriétés que nous expliquons ci-dessous.

5.2.3.4 Elimination de la récursivité à gauche.

Proposition 1 *Soit G une grammaire réduite, c'est-à-dire qui ne contient pas de terminaux inaccessibles ou improductifs,*
si G contient un non terminal récursif à gauche, alors elle n'est pas $LL(k)$.

Considérons par exemple la syntaxe concrète d'une expression arithmétique : La syntaxe concrète est la suivante :

$<$ *aexpr* $>$::= $<$ *aexpr* $><$ *opadd* $><$ *aterm* $>$
| $<$ *aterm* $>$

$<$ *aterm* $>$::= $<$ *aterm* $><$ *opmult* $><$ *aunary* $>$
| $<$ *aunary* $>$

$<$ *aunary* $>$::= $<$*aélément*$>$
|− $<$*aélément*$>$

$<$*aélément*$>$::= $<$ *entier* $>$
| $<$ *id* $>$
| $<$ *lexpr* $>$
| ($<$ *aexpr* $>$)

Cette grammaire est récursive à gauche : *une grammaire est récursive à gauche si un non-terminal A se dérive strictement en une proto-phrase commençant par A :*

$$A \Rightarrow A\alpha$$

Or une grammaire récursive à gauche n'est pas $LL(k)$, quelque soit k. L'algorithme suivant [7] permet d'éliminer la récursivité à gauche, par exemple :
$A ::= A\alpha|\ \beta$ devient $A ::= \beta A'$, $A' ::= \alpha A'|\ \epsilon$

Algorithme 1

- *pré : G sans cycle, sans production vide*
- *post : G sans récursivité à gauche*
- *invariant :* $\forall(A_k ::= A_l\alpha)$, *si* $k < i$ *alors* $l > k$ *(les règles déjà traitées commencent par des grands terminaux)*

pour $i := 1 \ldots n$
  pour $j ::= 1 \ldots i-1$
    remplacer A_j *par sa définition dans* $A_i ::= A_j\gamma$
  remplacer $A_i ::= A_i\alpha|\beta$ *par* $A_i ::= \beta A'_i$, $A'_i ::= \alpha A'_i|\epsilon$

En appliquant cet algorithme, on obtient pour l'exemple cité plus haut :

$< aexpr > ::=< aterm > (< opadd >< aterm >)^*$

$< aterm > ::=< aunary > (< opmult >< aunary >)^*$

$< aunary > ::=<$ *aélément* $>$
  $|-$ *<aélément>*

<aélément>$::= < entier >$
  $| < id >$
  $| < lexpr >$
  $|\ (\ < aexpr >\ )$

En effet, regardons de plus près par exemple :

$$\underbrace{< aexpr >}_{A_i} := \underbrace{< aexpr >}_{A_i}\underbrace{< opadd >< aterm >}_{\alpha}|\underbrace{< aterm >}_{\beta}$$

Après les transformations de l'algorithme :

$$\underbrace{< aexpr >}_{A_i} ::= \underbrace{< aterm >}_{\beta} A'_i\ ,\ A'_i ::= \underbrace{< opadd >< aterm >}_{\alpha} A'_i\ |\ \epsilon$$

A'_i correspond en fait à $(\alpha)^*$

5.2.3.5 Elimination des conflits

Cependant, il arrive que certaines règles de grammaire soient construites telles que, à un moment donné de l'analyse, le parseur est face à plusieurs expansions possibles (JavaCC nous prévient dans de telles situations). Il faut alors indiquer au parseur ce qu'il doit faire pour fonctionner correctement : des renseignements sont ajoutés dans la grammaire(LOOKEHEAD).

Par exemple, la règle suivante n'est pas $LL(1)$: il ne suffit pas au parseur de lire le premier *token* pour pouvoir choisir la bonne production.

$< lexpr >::= < id >$
$\quad | < id >$ [$< aexpr >$]

C'est une grammaire $LL(2)$ puisqu'il faut regarder les 2 premiers symboles pour savoir quelle est la bonne production.

Concrètement, pour le parseur, on écrit :

$< lexpr >::=$ LOOKEHEAD(2)
$\quad < id >$
$\quad | < id >$ [$< aexpr >$]

Autre exemple :

$< bexpr > ::=< bterm > (< opor >< bterm >)^*$

$< bterm > ::=< bunary > (< opand >< bunary >)^*$

$< bunary >::=<$*bélément*$>$
$\quad |- <$*bélément*$>$

$<$*bélément*$>::= < booleen >$
$\quad | < id >$
$\quad | < equation >$
$\quad |$ ($< bexpr >$)

tel que $< equation >::=< aexpr >< oprel >< aexpr >$

En se référant à la syntaxe des < *aexpr* > définie au point 4.2.5.4 et celle des < *bexpr* > décrite ci-dessus, analysons le comportement du parseur face à l'expression $(x + y < 6\ and\ y)$:

1. "(" : la règle de grammaire donne déjà le choix entre deux productions
   - la < *aexpr* > de < *equation* >
   - (< *bexpr* >)

   Le parseur choisit par exemple la première possibilité.
2. < *id* > : correspond à un < *aelement* > de l'< *aexpr* > parenthésée, le parseur poursuit puisque jusqu'à présent, c'est syntaxiquement correct.
3. "+" : on se trouve toujours dans l'< *aexpr* >, on continue.
4. < *id* > : < *aterm* > suivant de l'< *aexpr* >, ça marche toujours.
5. "<" : erreur de syntaxe de l'< *aexpr* >, puisque celle-ci ne peut pas contenir cet opérateur. Le premier choix n'était donc pas le bon.
6. On fait marche arrière jusqu'à l'étape du conflit (dans ce cas, l'étape 1) pour essayer une autre possibilité. Il s'agit du processus de *backtracking*.

Ce processus agit sur les performances, on résout ce problème de la manière suivante :

< *lexpr* >::= < *booleen* >
LOOKEHEAD(< *aexpr* >)
| < *equation* >
| < *id* >
|(< *bexpr* >)

Le LOOKEHEAD(< *aexpr* >) est intégré dans la syntaxe pour que le parseur généré sache qu'il doit lire toute l' *aexpr* avant d'affirmer qu'il a affaire à une expression de type *équation*.

5.2.3.6 Création d'arbres syntaxiques

JavaCC a la capacité de créer des arbres syntaxiques. Dans notre cas, il va instancier des classes qui représentent, selon la syntaxe abstraite, des composants de l'algorithme tels que les instructions et assertions. (voir section 5.2.4)

En fait, le parseur est généré de telle sorte qu'il vérifie et reconnaît la syntaxe de l'expression entrée, et en fonction de la règle reconnue de la grammaire, il crée un objet syntaxique.

C'est dans un fichier ($.jj$) qui sera exécuté par JavaCC que la syntaxe est rédigée. Cette syntaxe assume tout ce qui a été expliqué jusqu'à présent dans cette section (pas de récursivité à gauche , introduction de LOOKEHEAD) tout en introduisant au fur et à mesure des actions permettant :

- de mettre le contenu d'un token dans une variable : la méthode *token.image.toString()* permet de prendre le contenu du token qui la précède et de le transformer en String.
- de créer des objets syntaxiques, classes correspondant à la syntaxe abstraite définies au point 5.2.4 : en fonction du type d'expression parsée et du contenu des tokens, on instancie l'objet syntaxique correspondant.
- de mettre l'objet syntaxique résultant du parsing dans une variable (pour pouvoir agir dessus ultérieurement).
- d'agir sur ces objets : mettre une valeur dans un champ par exemple.

Exemple :

```
ObU DeclVar() :
{
Variable v;
String n;
String t;
}
{
< var >< id > {n = token.image.toString();} : < type >{t = token.image.toString();
                                                    if (t.equals("integer"))
                                                      v = new VarEnt(n);
                                                    else
                                                      v = new VarBool(n);
                                                    return v;}
}
```

Cet exemple représente une partie d'un fichier.jj, qui va être exécuté par JavaCC.

Cette partie correspond à l'analyse syntaxique d'une déclaration de variable dont la règle syntaxique est

$$\textbf{var}< id > : < type >$$

Le parseur crée l'objet *VarEnt* ou *VarBool* en fonction du *String t* qui contient le contenu du *token* $< type >$.

5.2.4 Le module objets syntaxiques

Les instances des classes de ce module représentent les objets de la syntaxe abstraite, qui constituent les composants d'un algorithme.

Ces objets syntaxiques sont divisés en quatre catégories : les objets utilisés (*i.e :* les déclarations), les situations (*i.e :* les spécifications pré-post-inv), les instructions (*i.e :* le corps de l'algorithme) et les expressions (pour la condition d'arrêt et la construction des objets cités ci-dessus).

Le but de cette section est de montrer ces objets selon deux angles différents : d'une part, nous détaillerons les champs de ces objets (partie syntaxique) et d'autre part, nous parlerons des méthodes qui les concernent, c'est-à-dire de la sémantique de ces objets et des vérifications de cohérence interne.

5.2.4.1 Les objets utilisés

Ces objets sont créés quand le parseur reconnaît une expression de déclaration. Ils n'ont pas de structure récursive, ni même compliquée. Analysons la structure des classes qui représentent ces objets utilisés.

La classe abstraite ObU

Le champ représentant le **nom** de l'objet utilisé (identifiant) a un grand rôle puisque c'est grâce à lui que l'on va pouvoir accéder à l'objet souhaité. Celui-ci est placé dans la mémoire(*Hashtable*) dès sa construction et on y accède par la clé d'accès qui se trouve être son identifiant (voir 5.2.1 L'environnement).

Le champ booléen **init** permet de savoir si l'objet a déjà été initialisé (il est bien sûr initialisé à false) et le champ **firstinit**[2] contient la(les) valeur(s) de l'objet à l'initialisation.

La méthode *Object GetVal()* renvoie la(les) valeur(s) de l'objet et utilisée pour afficher les résultats à chaque étape de l'exécution. Tandis que la méthode *CheckInit()* est utilisée pour vérifier les préconditions d'initialisation [3].

Cette classe est étendue par les classes concrètes *Constante* et *Tableau* et la classe abstraite *Variable* qui est elle-même concrétisée par les classes *VarEnt* et *VarBool.*

Les classes VarEnt et VarBool

[2] sera utilisé pour implémenter la sémantique de 4.19

[3] Implémentation de la relation 4.18

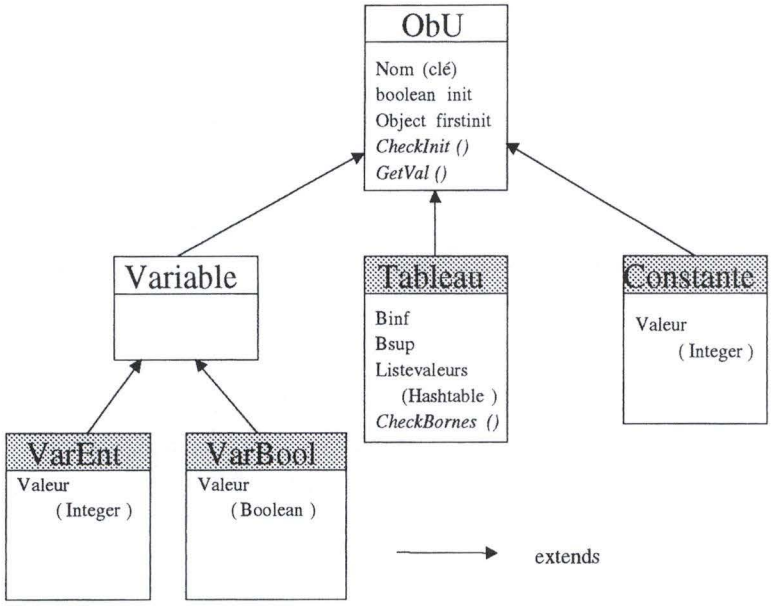

FIG. 5.2 – Classes représentant les objets utilisés

La classe *VarEnt/ VarBool* contient le champ concernant la **valeur** (*Integer/ Boolean*) de la variable, ce champ n'est bien entendu pas initialisé lors de la déclaration.

La méthode *CheckInit()* est implémentée au niveau de la classe *Variable*, et la méthode *Object GetVal()* est implémentée dans les classes concrètes.

La classe Tableau

Pour la classe *Tableau*, on a besoin de champs indiquant les **bornes** du tableau et d'un champ pointant vers **les valeurs** de celui-ci. Ce dernier est représenté par une *Hashtable* dont les clés sont des *Integer* qui pointent vers des objets *VarEnt*.

La méthode de vérification *CheckBornes()* s'assure de la cohérence des bornes du tableau initialisé. Dans le cas favorable, elle initialise les objets indicés et les met dans un nouvelle *hashtable* ayant comme clé d'accès les indices.

Les méthodes *CheckInit()* et *Object GetVal()* sont implémentées à ce niveau.

La classe Constante

Quant à la classe *Constante*, une instance de cette classe contiendra la *valeur* de la constante représentée.

Les méthodes *CheckInit()* et *Object GetVal()* sont triviales.

5.2.4.2 Les expressions

Ces objets syntaxiques plus complexes représentent de véritables arbres. De plus, ils *implémentent* selon le cas, l'interface *Aexpr* ou l'interface *Bexpr*.

Analysons plus précisément ces classes et interfaces et observons toutes les relations impliquées.

Les interfaces Aexpr et Bexpr

Commençons par les interfaces, celles-ci n'ont, par définition, que des méthodes abstraites :

- *Aexpr* possède la méthode abstraite *int Calculer(Hashtable h)* dont la signature correspond exactement à celle de la sémantique dénotationnelle

$$\mathcal{A} : \{aexpr\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{Z}$$

  C'est en fonction du paramètre h qui correspond à l'environnement (e et s) qu'on évalue l'*aexpr* et qu'on renvoie le résultat, un entier $\in \mathbb{Z}$.

- *Bexpr* possède la méthode abstraite *boolean Evaluer(Hahstable h)* dont la signature correspond exactement à celle de la sémantique dénotationnelle

$$\mathcal{B} : \{bexpr\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

  C'est en fonction du paramètre h qui correspond à l'environnement (e et s) qu'on évalue la *bexpr* et qu'on renvoie le résultat, un booléen $\in \mathbb{B}$.

En donc, toute classe qui *implémente* une interface *Aexpr*/*Bexpr* correspond à une expression arithmétique/booléenne et doit avoir une méthode implémentée de *Calculer(h)*/*Evaluer(h)*. Celle-ci sera la traduction directe d'une des relations décrites dans le chapitre 4 qui concerne la sémantique dénotationnelle.

La classe Expr

La classe abstraite *Expr* possède les méthodes abstraites dont les 1 et 2 correspondent aux vérifications de l'"état cohérent" [4] tandis que les 3 et 4 sont nécessaires pour les vérifications de complétude.

1. *CheckExist(Hashtable h)* : cette méthode vérifie que tous les éléments de l'expression existent bien, c'est-à-dire que les Lexpr apparaissant dans l'expression doivent correpondre à des objets utilisés existant dans la hashtable h.

[4] voir la sémantique des vérifications de cohérence de la section 4.2.2.

2. *CheckInit(Hashtable h)* : cette méthode vérifie que tout est bien initialisé dans l'expression.
3. *PutCompl(Hahstable h)* : cette méthode met dans un vecteur tous les objets mentionnés dans l'expression, ce vecteur sera utilisé pour les vérifications de complétude.
4. *Vector GetCompl()* : cette méthode renvoit le vecteur comprenant la listes des objets mentionnés dans une expression.

Ce sont les classes *Lexpr*, *Equation*, *CompExpr*, *Entier* et *Booleen* qui *étendent* la classe *Expr*.

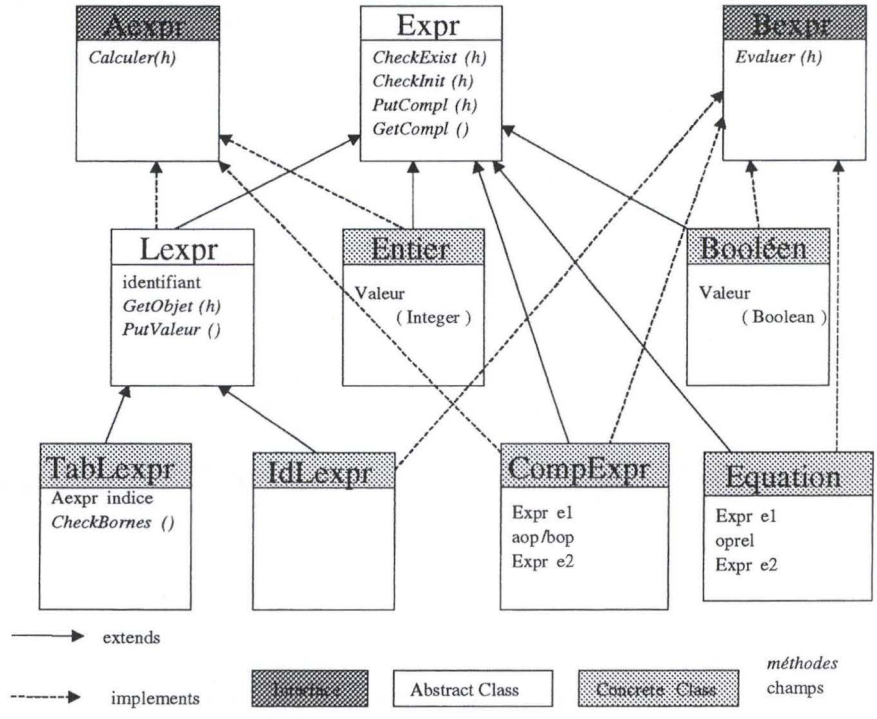

FIG. 5.3 – Classes et interfaces représentant les expressions

La classe Lexpr

Lexpr est une classe abstraite qui *implémente Aexpr* et qui est concrétisée par les classes *IdLexpr* et *TabLexpr*.

Lexpr contient un champ pour la clé d'accès à la mémoire, **key** et une méthode abstraite *Variable GetObjet()* qui retourne l'objet utilisé lui correspondant(4.1 et 4.2.2 de la sémantique dénotationnelle).

La méthode *int Calculer(h)* de *Aexpr* est implémentée ; elle correspond à la relation 4.5 de la sémantique.

La méthode *CheckInit(h)* de *Expr* peut également être implémentée à ce niveau.

La classe IdLexpr

IdLexpr pouvant être non seulement un entier mais aussi un booléen, *implémente* aussi *Bexpr*.

Dans cette classe, toutes les méthodes de *Expr* (sauf *CheckInit(h)* implémentée un niveau plus haut) et *Bexpr* (4.9) et *Lexpr* (4.1) sont donc implémentées.

Le champ de cette classe est la **clé** (ou nom) qui se trouve dans la *Lexpr* qu'elle étend.

La classe TabLexpr

Quant à *TabLexpr*, elle n'implémente pas la *Bexpr*, puisque le logiciel est réalisé pour un cadre de travail où on n'utilise que des tableaux d'entiers.

Les méthodes implémentées sont celles de la classe abstraite *Expr*(sauf *CheckInit(h)*) et celles qui correspondent à la relation (4.2.2) de la sémantique dénotationnelle (la méthode abstraite *GetObjet(h)* de *Lexpr*).

La relation 4.5 est implémentée au niveau de la *Lexpr*.

La classe *TabLexpr* a deux champs
- celui qui représente la **clé** ou le nom qui se trouve dans la *Lexpr* qu'elle étend,
- et celui qui représente l'**indice** (qui se trouve entre les crochets de la *Lexpr*)

La classe Equation

Une instance de la classe *Equation* représente une équation, elle *implémente* la méthode *Evaluer(h)* (4.12 et 4.11) de l'interface *Bexpr*, ainsi que toutes les méthodes abstraites de *Expr*.

Il peut s'agir d'équations ou inéquations entre expressions arithmétiques, ou bien d'équations entre expressions logiques. Cette distinction se fait déjà au niveau de la syntaxe.

Les champs de cette classe sont
- une *Expr* dont l'instance représente l'**expression** à gauche dans l'équation,
- un *String* qui contient l'**opérateur** de l'équation
- et une *Expr* dont l'instance représente l'**expression** à droite de l'opérateur

La classe CompExpr

La classe concrète *CompExpr* permet la représentation d'expressions composées, c'est-à-dire d'expressions contenant des opérateurs arithmétiques ou logiques. Cette classe *implémente* donc toutes les méthodes de *Expr*,*Aexpr* (4.6) et*Bexpr* (4.10) [5].

C'est l'analyse syntaxique qui vérifie déjà que les expressions qui entourent un opérateur arithmétique sont bien des expressions arithmétiques et il fait de même pour les opérateurs logiques.

Les champs de cette classe sont les mêmes que pour la classe *Equation*.

Les classes Entier et Booleen

Les instances des classes *Entier* et *Booleen* représentent des entiers ou des booléens élémentaires. Pour elles, les méthodes abstraites de *Expr* sont triviales et la méthode *Calculer(h)* (4.4) /*Evaluer(h)* (4.8) pour la classe *Entier*/*Booleen* renvoie simplement la valeur de l'entier/booléen représenté.

Pour chacune de ces deux classes, le champ est celui de la **valeur**.

Pour concrétiser ce qu'on a vu jusqu'à présent dans cette section, prenons un exemple à l'aide d'une représentation de l'environnement. Imaginons que les objets x et y sont initialisés et qu'on veut exécuter l'expression $x + y$:

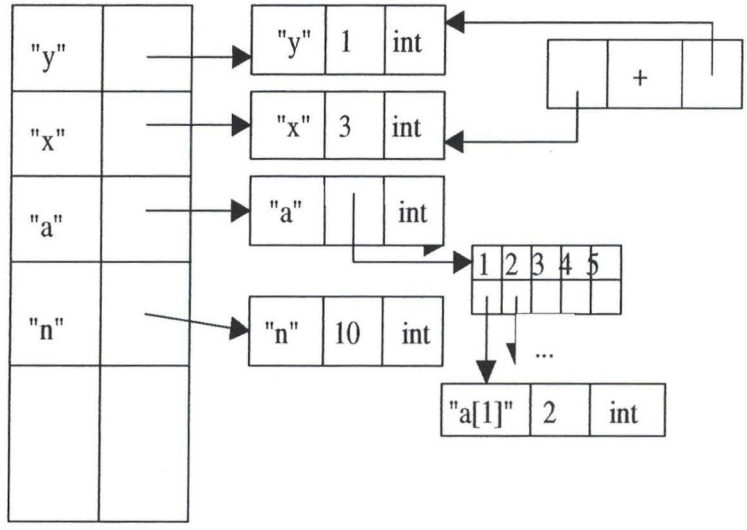

FIG. 5.4 – Exécution de l'évaluation d'une expression dans l'environnement h

[5]insistons encore sur l'évaluation de gauche à droite des expressions de disjonction et de conjonction.

5.2.4.3 Les situations

Etant donné que les *Bexpr étendent* les objets *Situation*, l'objet *Situation* est construit comme **interface**.

On considère donc que les classes abstraites *BFonct* et *Quantif implémentent* l'interface *Situation*.

Analysons ces classes et interfaces :

L'interface Situation

Elle contient les méthodes suivantes :

1. la méthode abstraite *boolean Evaluer(Hashtable h)* dont la signature correspond à celle de la sémantique dénotationnelle

$$\mathcal{S} : \{situation\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{B}$$

2. la méthode abstraite *CheckExist(Hashtable h)* : cette méthode vérifie l'*état cohérent* quand l'utilisateur enregistre une situation.
3. la méthode abstraite *PutCompl(Hashtable h)* : cette méthode met dans un vecteur tous les objets mentionnés dans l'assertion considérée.
4. la méthode abstraite *Vector GetCompl()* : cette méthode renvoie un vecteur qui contient tous les objets mentionnés dans une assertion.

Cette interface est *étendue* par l'interface *Bexpr* qu'on a déjà détaillé dans la partie concernant les expressions ; l'interface *Situation* est en plus *implémentée* par les classes concrètes *SitBexpr*, *ComSit*, *Domaine* et les classes abstraites *BFonct*, *Quantif*.

La classe concrète Domaine

Cette classe contient les champs représentant les éléments suivants :
- l'**identifiant** borné à gauche et à droite
- la **borne inférieure** qui est une *aexpr*
- la **borne supérieure** qui est une *aexpr*

Les méthodes *CheckExist(h)*, *PutCompl(h)* et *GetCompl()* sont implémentées, ainsi que la méthode *Evaluer(h)* qui correspond exactement à la sémantique de 4.17.

La classe concrète SitBexpr

Cette classe contient un champ correspondant à l'identifiant qui se trouve à gauche de l'équation(ou implication) et un champ qui représente la situation avec quantificateur qui se trouve à droite.

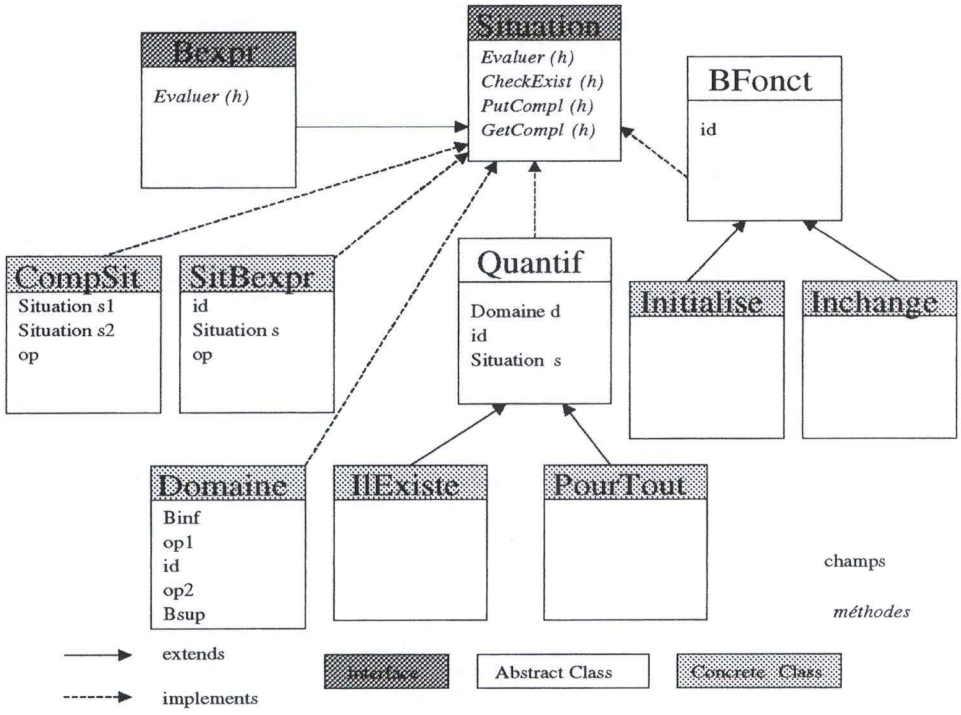

FIG. 5.5 – Classes et interfaces représentant les situations

Toutes les méthodes de l'interface sont implémentées. En particulier, la méthode *Evaluer(h)* est la traduction même de la relation 4.22.

La classe concrète CompSit

Cette classe contient deux champs représentant les situations qui composent cette situation, et un opérateur booléen.

Toutes les méthodes de l'interface sont implémentées. En particulier, la méthode *Evaluer(h)* est la traduction même de la relation 4.23.

La classe abstraite BFonct

Cette classe abstraite contient un champs identifiant qui servira à préciser sur quel objet la fonction booléenne est appliquée. Cette classe implémente déjà les méthodes *CheckExist(h)*, *PutCompl(h)*et *GetCompl()*. La méthode *Evaluer(h)* est implémentée dans les classe concrètes (*i.e :* les classes *Initialisé* et *inchangé*).

La classe concrète Initialisé

Cette classe implémente juste la fonction *Evaluer(h)* qui correspond à la relation

4.18 dans la sémantique dénotationnelle.

La classe concrète Inchangé

Cette classe implémente juste la fonction *Evaluer(h)* qui correspond à la relation 4.19 dans la sémantique dénotationnelle.

La classe abstraite Quantif

Cette classe abstraite possède les champs correspondant à
- l'identifiant "quantifié"
- le domaine sur lequel il est quantifié
- la situation qui suit cette quantification

Elle implémente la méthode *CheckExist(h)*, *PutCompl(h)* et *GetCompl()* à ce niveau.

La classe concrète IlExiste

Cette classe implémente la relation 4.21 de la sémantique dénotationnelle (*i.e :* la méthode *Evaluer(h)* de l'interface *Situation.*

La classe concrète PourTout

Cette classe implémente la relation 4.20 de la sémantique dénotationnelle (*i.e :* la méthode *Evaluer(h)* de l'interface *Situation.*

5.2.4.4 Les instructions

La classe abstraite Instruction

Cette classe contient les méthodes abstraites de cohérence :
- *CheckExist(Hashtable h)* qui vérifie si les objets utilisés dans l'instruction ont bien été déclarés et qui assure que les parties de droite des affectations sont bien définies.
- *CheckType(Hahstable h)* qui vérifie si le type de la partie de gauche correspond au type de la partie de droite des affectations.
- *CheckUtil(Hashtable h, Vector v)* utilisée dans le cas d'une instruction de clôture pour s'assurer de l'utilité de l'instruction en fonction de la postcondition (dans le *Vector v*).

Elle contient également la méthode abstraite d'exécution : *Executer(Hashtable h)* qui consiste à réaliser les affectations, *i.e :* mettre la valeur de l'expression de droite dans l'objet correspondant à l'expression de gauche. La signature de la sémantique dénotationnelle correspondante est la suivante :

$$\mathcal{I} : \{instr\} \rightarrow \mathbb{E} \rightarrow \mathbb{S} \rightarrow \mathbb{S}$$

Cette classe est concrétisée par les classes *Affectation* et *Conditionnelle* :

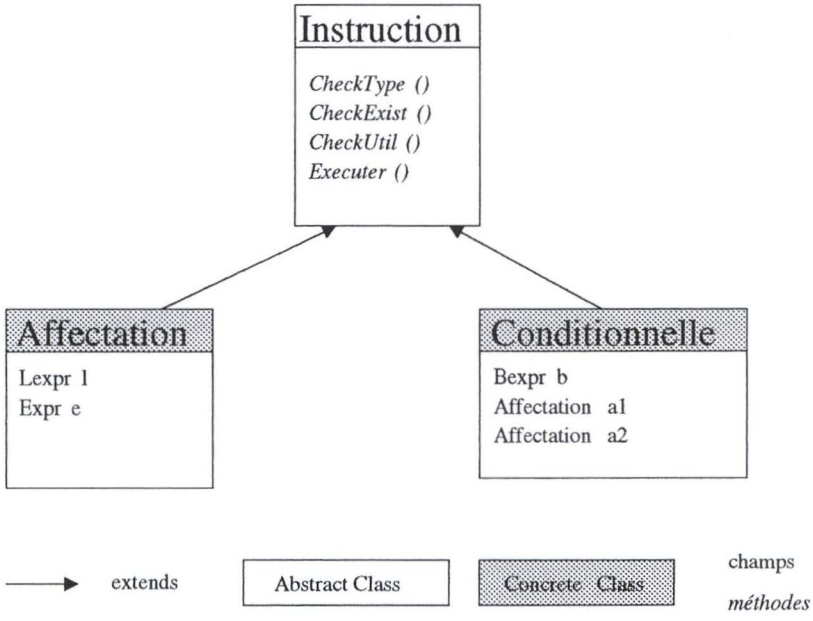

FIG. 5.6 – Classes représentant les instructions

La classe concrète Affectation

Elle implémente toutes les méthodes abstraites décrites ci-dessus [6] ; la méthode *Executer(h)* correspond à la relation 4.13 de la sémantique.

La classe concrète Conditionnelle

Elle implémente toutes les méthodes abstraites décrites ci-dessus [7] ; la méthode *Executer(h)* correspond à la relation 4.14 de la sémantique.

[6] selon la sémantique des vérifications de cohérence décrites dans la section 4.2.2.
[7] selon la sémantique des vérifications de cohérence décrites dans la section 4.2.2.

Chapitre 6

Critiques et objectifs ultérieurs

Le logiciel tel qu'il est présenté et réalisé permet déjà la construction de certains algorithmes, mais étant donné que de nombreuses primitives ne sont pas encore programmées, un grand nombre de problèmes ne peut être testé.

Pour le rendre effectif, il suffit donc d'étendre la syntaxe c'est-à-dire de permettre des nouveaux mots et de leur attribuer une sémantique définie par une primitive.

Il ne serait pas négligeable de pouvoir prouver la terminaison de l'algorithme construit. Il s'agit quand même d'une des étapes fondamentales de la méthode de programmation décrite dans le chapitre 2[1].

Le fait de pouvoir stocker les données serait un avantage considérable ; ainsi, l'étudiant pourrait revenir à son travail interrompu ou à un problème résolu.

Lorsque les concepts de cette méthode de programmation sont bien assimilés par l'étudiant, il serait intéressant que le logiciel permette la décomposition en sous-problèmes.
Exemple : afin de résoudre le problème qui consiste à trier un tableau par ordre croissant, il faut d'abord résoudre le problème de la recherche du maximum dans le tableau.

Comme le niveau de rigueur des assertions mathématiques est la faiblesse générale des étudiants, on pourrait considérer une autre forme de langage : exprimer les situations sous forme de schéma [3] et [4]. De cette manière, on se rend compte que l'étudiant fait beaucoup moins d'erreurs. En effet, ce n'est pas la compréhension du problème qui implique tant d'erreurs mais la traduction en mathématique. On pourrait alors vérifier la cohérence entre ces deux formes d'expression de situations, et l'étudiant comprendrait ainsi directement à quel style d'erreur il est confronté.

[1] à défaut de le prouver, on pourrait déjà tester qu'un "variant" majoré croît strictement.

Enfin, dans des perspectives plus difficiles et ambitieuses, l'analyse statique des algorithmes permettrait que toutes les corrections se fassent pas à pas dans la démarche de l'étudiant. Cette méthode utiliserait des techniques de preuves formelles et d'interprétation abstraite. Quant au logiciel décrit dans ce mémoire, il est conçu de telle sorte que toutes les vérifications de cohérence entre les spécifications et les instructions se font à l'exécution(et donc de façon dynamique). Seules les cohérences internes aux spécifications ou aux instructions se font pas à pas dans la démarche.

Pour terminer, on pourrait étendre cette idée de logiciel : un logiciel ferait preuve d'une grande efficacité s'il arrivait à éclairer le concept de la récursivité, qui est l'obstacle le plus persistant et habituel chez les étudiants.

Conclusion

La base de ce travail fut la compréhension de la méthodologie en programmation. Cela m'a permis d'approcher cette discipline non plus avec les yeux d'une étudiante devant l'apprendre, mais en tant que personne désirant en comprendre l'intérêt tout en réfléchissant à une pédagogie pour l'enseigner.

De plus, je me suis forgée une expérience au niveau de la conception d'un logiciel. En effet, j'ai pu mettre en pratique certains concepts purement théoriques étudiés en première licence. J'ai également appliqué des notions de méthodologie de développement de logiciel. Enfin, ce travail m'a permis de me familiariser un peu plus avec le langage de programmation java.

La première partie du travail consistait simplement à réfléchir à une interface : connaissant la méthode étudiée en première candidature, je devais imaginer un scénario assez souple qui permettrait aux étudiants de mieux "sentir" la méthode.

Dans la seconde étape, tout à fait indépendante de la première, il s'agissait de fixer la syntaxe utilisable avec le futur logiciel, et de créer un analyseur syntaxique. Parallèlement, j'ai construit des classes java, objets syntaxiques qui représentent les composants d'un algorithme (spécifications et instructions) selon la syntaxe abstraite.

Ensuite, il fallait implémenter l'interface et toutes les méthodes nécessaires aux vérifications de cohérence et de complétude. Il restait ainsi à assembler le tout pour que finalement, l'interface fasse appel à l'analyseur syntaxique et aux fonctions de vérifications, tout en utilisant les objets syntaxiques.

La suite du travail est encore longue mais cette formation très rigoureuse de l'algorithmique est très riche, notamment par la prépondérance accordée à la rigueur du raisonnement, ce serait dommage de laisser tomber cette approche sous prétexte qu'elle pose un problème d'intuition chez les étudiants.

Bibliographie

[1] J.ARSAC, Vous avez dit algorithmique?, Actes du deuxième colloque francophone sur la didactique de l'informatique, FUNDP, 1990.

[2] P.A.DE MARNEFFE, Enseigner l'algorithmique, Actes du deuxième colloque francophone sur la didactique de l'informatique, FUNDP, 1990.

[3] M.DERROITE, B.LE CHARLIER, Un système d'aide à l'enseignement d'une méthode de programmation, Actes du premier colloque francophone sur la didactique de l'informatique, EPI, ISSN 0758-590X, cours suivi en 1989.

[4] O.DETIÈGE, X.ZÉBIER, Définition et implémentation d'un langage graphique pour la description d'assertions, mémoire, FUNDP, 1998.

[5] B.LE CHARLIER, Introduction à la programmation, cours de première candidature en sciences mathématiques, FUNDP, cours suivi en 1994.

[6] B.LE CHARLIER, Paradigmes de programmation, cours de première licence et maîtrise en informatique, FUNDP, cours suivi en 1998.

[7] P.Y.SCHOBBENS, Syntaxe et sémantique, cours de première licence et maîtrise en informatique, FUNDP, cours suivi en 1998.

[8] B.ECKEL, Thinking in Java,Prentice Hall, 1998.

[9] http ://java.sun.com/products/jdk1.2.2/docs/api/

[10] Borland JBuilder3, Version 3.00, Inprise Corporation.

[11] Inria,Javacc Documentation.
URL :http ://falconet.inria.fr/~ java/tools/JavaCC/examples/

[12] Javacc Documentation.
URL :http ://www.cs.um.edu.mt/~ java/javacc-docs/DOC/index.html