Département
Informatique

Annexes

Compilateur de programmes CSP en JAVA

Benjamin Le Bozec

24 juin 2005

Encadrant : M. Olivier ROUX
Responsable : M. Sébastien FAUCOU

Stage du 10 avril au 18 juin 2005

version 1.1

Suivi du document

version 1.0 - 20/06/2005 - Version initiale

version 1.1 - 24/06/2005 - Modifications orthographiques

Table des matières

1 Complément sur le langage CSP

1.1 Détails sur les commandes

La commande nulle est représentée par le mot clé `skip`. Elle n'a pas d'utilité particulière, n'effectue aucune action, et ne peut échouer.

La commande d'affectation se présente sous la syntaxe suivante :

```
variableCible := expressionSource
```

où `variableCible` et `expressionSource` peuvent être des signaux, ou des variables. Exemples :

```
a := 5
b := a
x := p(a,b)
p(a,b) := x
p(a,b) := p(b,a)
```

1.2 Compléments sur les apports faits au langage

1.2.1 Syntaxe des déclarations

Les déclarations de variable n'ayant pas été traitées dans l'article de HOARE, elles ont été rajoutées sous la forme suivante :

- `var1,var2,... : type`
- ou avec initialisation : `var1,var2,... : type := valeur_initiale`

De plus, HOARE stipule dans son article que les variables désignant un signal doivent être typées à la première affectation. Mais une telle règle complique nécessairement le compilateur d'une part, et d'autre part peut poser des problèmes de *correspondance des types* lors d'une communication inter-processus, comme le montre l'exemple suivant :

```
[ proc1 :: proc2!p() || proc2 :: proc1?x ]
```

Ici, la variable `x` n'étant pas typée, il est impossible de savoir si la communication est valide ou non. Ainsi, pour palier à ce problème, la déclaration des variables devant contenir un signal est imposée, et se présente sous la forme :

```
variable : constructeur(type1, type2, ...) := constructeur(valeur1, valeur2, ...)
```

(l'initialisation n'est pas obligatoire)

Par exemple :

- `x : p(integer, integer) := p (3, 5)`
- `x : p( q(string, string), integer)`
- `x : p()`

1.2.2 Déclaration de processus

Afin de d'augmenter la lisibilité du code, une notion de *déclaration de processus* a été rajoutée au langage. Elle permet de nommer une liste de commandes en vue de son utilisation comme processus, et se présente sous la syntaxe suivante :

```
IDENTIFIANT == liste de commandes
```

l'identifiant pouvant alors être utilisé dans les commandes parallèles.

Les liste de commandes ainsi déclarées sont aussi appelées **macros**. On considère alors un programme comme étant une liste de macros, l'un des macros devant avoir l'identifiant `MAIN`, définissant ainsi la liste de commandes a exécuter au lancement du programme.

L'exemple suivant illustre ces différentes notions :

```
PRODUCTEUR == nb : integer := 0; consommateur!nb ()

CONSOMMATEUR == nb : integer; producteur?nb ()

MAIN == [ producteur :: PRODUCTEUR || consommateur :: CONSOMMATEUR]
```

1.2.3 Déclaration des constantes

Les constantes permettent d'associer à un identifiant une valeur qui peut être un nombre ou une chaîne de caractères. Elles se présentent sous la forme :

```
define IDENTIFIANT VALEUR
```

et doivent être déclarée au début du programme, c'est-à-dire avant la déclaration du premier macro.

1.3 Grammaire du langage

1.3.1 Notations utilisées

Les notations utilisées dans la suite pour exprimer la grammaire sont relativement classiques :

- les règles sont de la forme : `règle : := ensemble de substituts ;` ;
- `(substitut) ?` indique que le substitut peut être présent ou absent ;
- `(substitut)*` indique que le substitut peut apparaître plusieurs fois, ou pas du tout ;
- `(substitut)+` indique que le substitut doit apparaître au moins une fois ;
- `substitut1 | substitut2` indique une alternative entre les deux substituts ;
- `substitut1 substitut2` indique que `substitut1` doit être rencontré, puis `substitut2`.

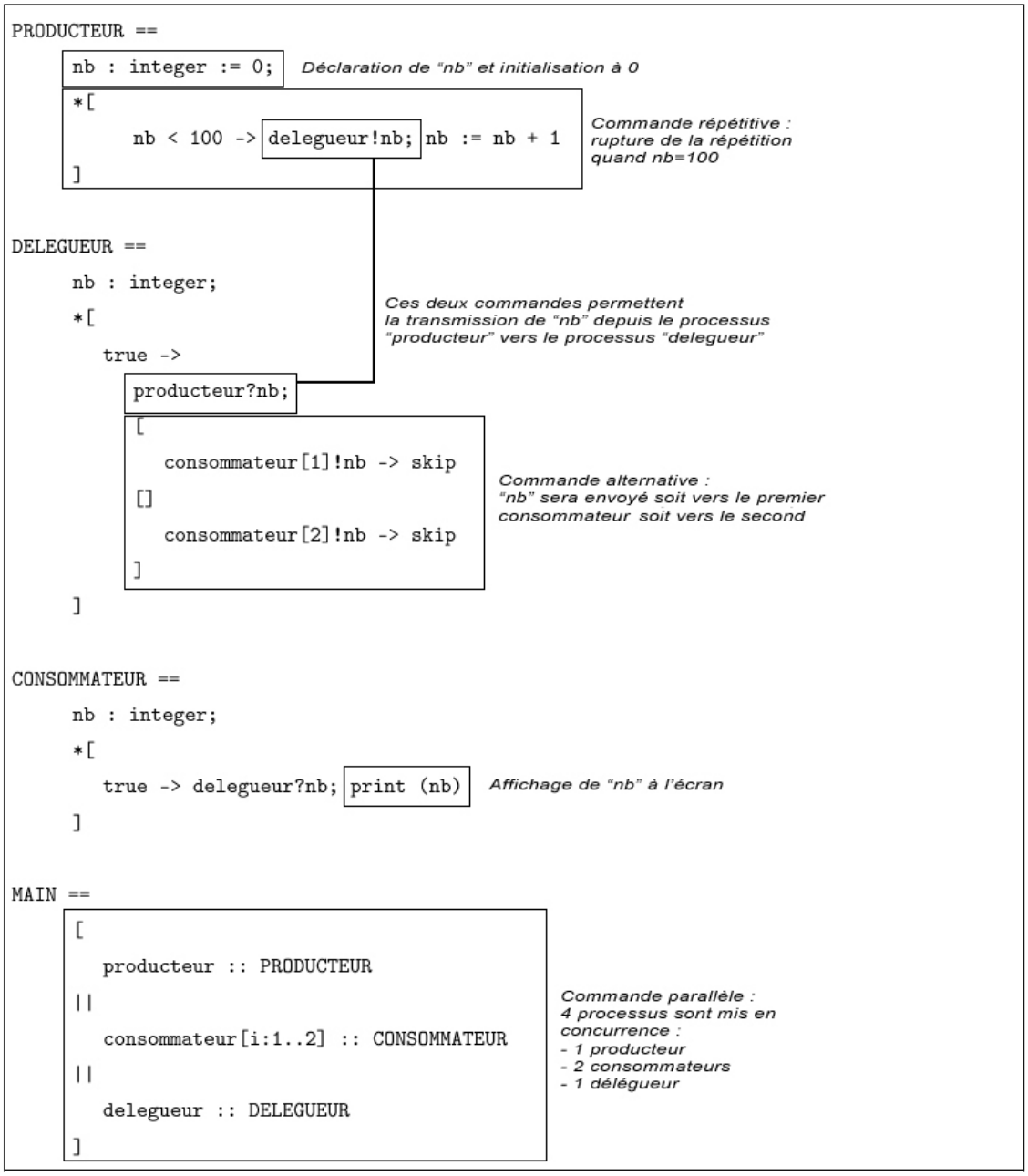
1.3.2 Grammaire

```
programme ::= ( declarationConstante )* ( declarationMacro )+ EOF;
declarationConstante ::= "define" identifier valeur;
string ::= STRING_LITERAL;
valeur ::= nombre | string | "true" | "false";
valeurTab ::= variable "[" expressionSimple ( "," expressionSimple )* "]";
variable ::= IDENTIFIANT;
typeBase ::= "integer" | "string" |"boolean";
typeTab ::= "[" simpleRange ( "," simpleRange )* "]" typeBase;
type ::= typeBase | typeStructuree | typeTab;
typeStructuree ::= constructeur "(" ( type ( "," type )* )? ")";
nomProcessus ::= IDENTIFIANT;
nombre ::= ENTIER;
declarationMacro ::= idMacro "==" listeCommandes;
idMacro ::= IDENTIFIANT;
listeCommande ::= commande ( ";" commande )*;
commande ::= commandeSimple | commandeStructuree;
commandeSimple ::= commandeNulle | declaration | commandePrint
                     | commandeSleep | commandeAffectation | commandeES;
commandeSleep ::= "sleep" "(" expressionSimple ")";
commandePrint ::= "print" "(" expressionSimple ")";
commandeNulle ::= "skip";
declaration ::= listeVariable ":" type ( ":=" expression )?;
listeVariable ::= variable ( "," variable )*;
commandeAffectation ::= variableCible ":=" expression;
variableCible ::= cibleArray | cibleStructuree | variable;
```

```
cibleArray ::= variable "[" expressionSimple ( "," expressionSimple )* "]";
cibleStructuree ::= constructeur "(" ( listeVariableCible )? ")";
constructeur ::= IDENTIFIANT;
listeVariableCible ::= variableCible ( "," variableCible )*;
expression ::= expressionStructuree | expressionTab | expressionSimple;
expressionTab ::= "[" simpleRange ( "," simpleRange )* "]" expressionSimple;
expressionStructuree ::= constructeur "(" ( listeExpression )? ")";
listeExpression ::= expression ( "," expression )*;
commandeES ::= commandeSortie | commandeEntree;
commandeSortie ::= processusCible "!" expression;
commandeEntree ::= processusCible "?" variableCible;
processusCible ::= nomProcessus ( "[" listeIndice "]")?;
listeIndice ::= expressionSimple ( "," expressionSimple )*;
commandeStructuree ::= commandeParallele | commandeAlternative | commandeRepetitive;
commandeParallele ::= "[" listeProcessus "]";
listeProcessus ::= processus ( "||" processus )*;
processus ::= labelProcessus "::" (listeCommande | idMacro );
labelProcessus ::= nomProcessus ( "[" listeIndiceLabel "]")?;
listeIndiceLabel ::= indiceLabel ( "," indiceLabel )*;
indiceLabel ::= expressionSimple | range;
simpleRange ::= expressionSimple ".." expressionSimple;
range ::= variable ":" expressionSimple ".." expressionSimple;
commandeRepetitive ::= "*" commandeAlternative ;
commandeAlternative ::= "[" listeCommandeGardee "]";
listeCommandeGardee : commandeGardee ( "[]" commandeGardee )*;
commandeGardee ::= garde "->" listeCommande;
garde ::= expressionSimple ";" commandeES | commandeES | expressionSimple;
listeDeclaration ::= declaration (";" declaration )*;
expressionSimple ::= exprBool ( ( "and" | "or" ) exprBool )*;
exprBool ::= expressionOp ( ( "=" | "/=" | ">=" | "<=" | ">" | "<" ) expressionOp )*;
expressionOp ::= expr ( ( "+" | "-") expr )*;
expr ::= atom ( ( "*" | "/" | "mod" ) atom )*;
atom ::= valeurTab | valeur | variable | "(" expressionSimple ")" | "-" atom;
```

1.4 Exemples de programmes CSP

1.4.1 Exemple détaillé, cas producteur/consommateur



1.4.2 Producteurs/consommateurs avec tampon

Rappelons le principe de ce paradigme : des producteurs produisent des éléments puis vont les placer dans une liste, tandis que des consommateurs vont chercher dans cette même liste des éléments à consommer. C'est ce qui se produit par exemple lorsqu'une imprimante est reliée à plusieurs ordinateurs : chaque utilisateur met dans une liste commune ses documents à imprimer, puis l'imprimante vient les *consommer* un par un.

Dans le programme qui suit, des constantes ont été définies afin de permettre une meilleure manipulation :

- NP correspond au nombre de producteurs ;
- NC au nombre de consommateurs ;
- et TL à la taille de la liste intermédiaire.

```
define NP 2
define NC 2
define TL 10

PRODUCTEUR ==
     nb : integer := 0; message : string;
     *[
        true ->
           message := "objet num " + nb;
           liste!prod(message);
           print ("production " + i + " : " + message + "\n");
           nb := nb + 1
     ]

LISTE ==
     tab : [0..TL-1] string := [0..TL-1] "";
     used,mark : integer := 0;
     message : string := "";
     *[
        (i:1..NP) used < TL; producteur[i]?prod(tab[(mark + used) mod TL]) ->
           used := used + 1
     []
        (i:1..NC) used > 0; consommateur[i]!cons(tab[mark]) ->
           mark := (mark + 1) mod TL;
```

```
            used := used - 1
      ]

CONSOMMATEUR ==
      message : string;
      *[
         true ->
            liste?cons(message);
            print ("consommateur " + i + " : " + message + "\n")
      ]

MAIN ==
      [
         producteur[i:1..NP] :: PRODUCTEUR
      ||
         consommateur[i:1..NC] :: CONSOMMATEUR
      ||
         liste :: LISTE
      ]
```

1.4.3 Lecteurs/écrivains avec priorité aux écrivains

Le principe est le suivant : une ressource est partagée entre des lecteurs (des processus accédant la ressource en lecture), et des écrivains (accédant la ressource en écriture, c'est-à-dire dans le but de la modifier). Un contrôleur permet alors de gérer les accès, en donnant la priorité aux écrivains, souvent beaucoup moins nombreux que les lecteurs.

Ici, la constante NE correspond au nombre d'écrivains, tandis que NL correspond au nombre de lecteurs. La ressource est un entier que les écrivains incrémentent et que les lecteurs affichent à l'écran.

Notons aussi la présence de la commande `sleep`, permettant d'une part de simuler un accès long à la ressource, et d'autre part de simuler des accès *peu fréquents*.

```
define NE 4
define NL 20

ECRIVAIN ==
   *[ true ->
      sleep (5000);
      ctrl!accesPrioritaire ();
      ctrl!acces ();
      ressource := ressource + 1;
      sleep (5000);
      ctrl!libere ()
   ]



LECTEUR ==
   *[ true ->
      ctrl!acces ();
      print ("ressource : " + ressource + "\n");
      ctrl!libere ()
   ]

CTRL ==
   ne : integer := 0; nl : integer := 0;
   *[
```

```
    (i:1..NL) ne=0; lecteur[i]?acces() -> nl := nl + 1
  []
    (i:1..NL) lecteur[i]?libere () -> nl := nl - 1
  []
    (i:1..NE) ecrivain[i]?accesPrioritaire () -> ne := ne + 1
  []
    (i:1..NE) nl=0; ecrivain[i]?acces () -> ecrivain[i]?libere (); ne := ne - 1
  ]

MAIN ==
  ressource : integer := 0;
  [
    ecrivain [i:1..NE] :: ECRIVAIN
  ||
    lecteur [i:1..NL] :: LECTEUR
  ||
    ctrl :: CTRL
  ]
```

1.4.4 Chien de garde

Le programme suivant présente un cas d'émetteur/récepteur gérer par un chien de garde : un émetteur envoie des messages à un récepteur, et de temps en temps on simule le blocage de l'émetteur (simulation de la perte d'un paquet par exemple). Ainsi, si le récepteur n'a pas reçu de messages depuis 4 secondes, le chien de garde se charge de déclencher une erreur afin de relancer la communication.

```
EMETTEUR ==
   message : string; cpt : integer := 0;
   * [ true ->
      cpt := cpt + 1;
      message := "message " + cpt;
      recepteur!message;
      sleep(1000);
      [
         (cpt mod 5) = 0 -> recepteur?relance()
      []
         (cpt mod 5) /= 0 -> skip
      ]
   ]

RECEPTEUR ==
   message : string; alive : boolean := true;
   *[
      alive; emetteur?message -> dog!ack(); print (message + "\n")
   []
      alive; dog?erreur() -> print("erreur\n"); emetteur!relance(); dog!ackErreur ()
   ]

DOG ==
   delai : integer := 4;
   *[
      recepteur?ack() -> delai := 4
   []
      horloge?bip() -> delai := delai -1
   []
      delai = 0 -> recepteur!erreur(); recepteur?ackErreur(); delai := 4
   ]
```

```
HORLOGE ==
   *[true -> dog!bip();sleep(1000)]

MAIN ==
   [
      emetteur :: EMETTEUR
   ||
      recepteur :: RECEPTEUR
   ||
      dog :: DOG
   ||
      horloge :: HORLOGE
   ]
```

2 Le package csp_runtime

CspThread

Correspond à la super classe de tout processus CSP.

Attributs

- `boolean mustStop`

  Indique au processus s'il doit s'arrêter ou non (utilisé par l'interface)

Méthodes

- `protected void input (CspThread cible, Signal signal)`

  correspond à une commande d'entrée. `cible` *et* `signal` *sont alors encapsulés dans un objet* `CommandeES` *et sont envoyés au contrôleur.*

- `protected void output (CspThread cible, Signal signal)`

  correspond à une commande de sortie.`cible` *et* `signal` *sont alors encapsulés dans un objet* `CommandeES` *et sont envoyés au contrôleur.*

- `protected void fireDeath ()`

  permet au processus d'indiquer qu'il a atteint la fin de son exécution. L'appel est simplement dérivé vers le contrôleur.

- `public void start()`

  lance le processus.

- `public boolean join ()`

  attend la fin du processus et retourne `vrai` *s'il a terminé avec succès,* `faux` *sinon*

- `protected void print (String message)`

  permet au processus d'afficher un message à l'écran. En réalité, cette méthode se charge juste de dispatcher le message vers les différents listeners venus se référencer.

- `protected boolean debug (int l1, int c1, int l2, int c2)`

  permet au processus d'indiquer la prochaine commande qu'il va exécuter. Le couple `(l1, c1)` *correspond aux coordonnées* `(ligne, colonne)` *du premier caractère de la commande dans le fichier source, tandis que* `(l2, c2)` *correspond au dernier caractère de la commande.*

CspData<T> implements Valeur

Englobe toute variable CSP, permettant ainsi de manipuler les variables en tant que référence et non plus en tant que valeur. L'utilisation des génerics permet ici d'englober n'importe quel type.

Attributs

- `public T value`

  Valeur associée à l'objet.

Méthodes

- `public CspData (T value)`

  Constructeur initialisant la valeur de l'objet.

- `public T get()`

  Retourne ***`value`***

- `public void set(T value)`

  Modifie la valeur de ***`value`***

- `public void copyFrom (CspData<T> d)`

  Récupère la valeur associée à ***`d`*** *et l'affecte à* ***`value`***

- `public boolean matches (CspData<?> d)`

  Vérifie si le type associé à l'objet ***`d`*** *est le même que celui de l'objet courant.*

CspArray<T> implements Valeur

Correspond à un tableau CSP. Permet la gestion d'indices avec bornes inférieure et supérieure.

Attributs

- `private Borne [] bornes`

  Liste des bornes (un objet ***`Borne`*** *est composée d'un attribut* ***`borneInf`*** *et d'un attribut* ***`borneSup`****)*

- `private CspData<?> [] tab`

  Tableau des valeurs. Un tableau unidimensionnel est utilisé et découpé en fonction des bornes.

Méthodes

- `public CspArray (T initialValue, Bornes ... bornes)`

  Construit un objet ***`CspArray`*** *en initialisant toutes les valeurs du tableau à* ***`intialValue`*** *et utilisant la liste d'indice définit par le tableau.*

- `public void set (T value, int ... index)`

  Affecte ***`value`*** *à l'élément décrit par la liste d'indices* ***`index`****.*

- `public T get (int ... index)`

  Retourne la valeur de l'élément décrit par la liste d'indices ***`index`****.*

- `public CspData<T> getData (int ... index)`

  Retourne l'objet ***`CspData`*** *associé à l'élément décrit par la liste d'indices* ***`index`****.*

- `public boolean matches (CspArray<?> tab)`

  Vérifie si le type associé au tableau ***`tab`*** *correspond au type de l'objet courant.*

- `public void copyFrom (CspArray<T> tab)`

Récupère toutes les valeurs du tableau `tab` *et les affecte aux éléments du tableau courant. Les deux tableaux doivent se correspondre (i.e.* `vrai` *doit être retourné par la méthode* `matches`*).*

Signal implements Valeur

Correspond à la notion de signal en CSP.

Attributs

- `private String constructeur`

  Constructeur associé au signal.

- `private Valeur [] data`

  Valeurs associée au signal.

Méthodes

- `public Signal (String constructeur, Valeur ... data)`

  Construit un nouvel objet `Signal` *avec le constructeur et les valeurs donnés.*

- `public boolean matches (Signal signal)`

  Vérifie si l'objet `signal` *correspond à l'objet courant (le constructeur doit être identique ainsi que le type des valeurs)*

- `public void copyFrom (Signal signal)`

  Récupère les valeurs de l'objet `signal` *et les affecte aux valeurs de l'objet courant. Une exception est levée si la méthode* `matches` *retourne* `faux`*.*

Select

Permet de simplifier l'accès au contrôleur pour le traitement d'une alternative.

Attributs

- `private Garde [] gardes`

  Liste des gardes associée à l'alternative à traiter.

Méthodes

- `public void addInput (int num, BoolExpr boolExpr, CspThread dest, Signal signal)`

  Référence une garde avec commande d'entrée. `boolExpr` *correspond à la partie booléenne (peut être égale à null),* `dest` *et* `signal` *correspondent quand à eux au paramètre de la commande d'entrée.* `num` *correspond quant à lui au numéro de la sélective, numéro qui sera retourné si la selective est choisie.*

- `public void addOutput (int num, BoolExpr boolExpr, CspThread dest, Signal signal)`

  Même chose que `addInput`*, mais avec une commande de sortie.*

- `addSimpleExprBool (int num, BoolExpr boolExpr)`

  Permet d'ajouter une garde composée simplement d'une partie booléenne. `num` *correspond au numéro de la sélective, numéro qui sera retourné si la sélective est choisie.*
- `public boolean select ()`

  Cette méthode est chargée d'envoyer au contrôleur l'ensemble des gardes précédemment référencée, puis retourne `vrai` *si une garde a été sélectionnée, ou* `faux` *si toutes les gardes étaient à* `faux`*.*
- `public int getChoice ()`

  Retourne le numéro de la selective choisie par le contrôleur, ou -1 si aucune sélective n'a été choisie.

Garde

Objet englobant les propriétés relatives à une garde. **Attributs**

- `public final BoolExpr boolExpr`

  Partie booléenne de la garde
- `public final CommandeES cmdES`

  Commande d'entrée/sortie associée à la garde
- `public boolean choosen = false`

  Booléen indiquant si la garde a été choisie ou non

CommandeES

Objet englobant les propriétés relatives à une commande d'entrée/sortie. **Attributs**

- `public final CspThread source`

  Processus ***émetteur*** *de la commande d'e/s*
- `public final CspThread dest`

  Processus ***destination*** *de la commande d'e/s*
- `public final IOType ioType`

  Type de la commande (entrée ou sortie, IOType étant une classe `enum`*)*
- `public final Signal signal`

  Le signal concerné par la commande (pouvant jouer le rôle de variable cible ou d'expression source)
- `public boolean success = false`

  Booléen indiquant si la commande d'entrée/sortie a échouée ou non

Méthodes

- `public void perform (CommandeES cmd`

  Effectue une transmission avec la commande donnée en paramètre.

- `public boolean matches (CommandeES cmd)`

  Vérifie si la commande d'entrée/sortie `cmd` *peut communiquer avec la commande courante*

interface BoolExpr

Interface permettant de déclarer une expression booléenne qui ne sera testée que lors de l'évaluation d'une garde. **Méthodes**

- `public void eval()`

  Cette méthode est appelée par le contrôleur pour récuperer la valeur de l'expression booléenne

interface Valeur<T>

Interface englobant tous les types susceptibles de contenir des valeurs, c'est-à-dire `CspData`, `CspArray`, et `Signal`. **Méthodes**

- `public boolean matches (T valeur)`

  Vérifie si l'objet `valeur` *correspond à l'objet courant*

- `public void copyFrom (T valeur)`

  Affecte à l'objet courant les valeurs de l'objet `valeur`

Controleur

Il est chargé de gérer les communications inter-processus. **Attributs**

- `public Semaphore mutex`

  Sémaphore permettant de ne gérer qu'une demande de communication à la fois

Méthodes

- `public void manageGardes (Garde [] gardes)`

  Évalue les gardes données en entrée selon les principes imposés par CSP

- `public void fireDeath (CspThread thread)`

  Permet de déclarer la fin du processus `thread`

- `public Garde [] filtreGardes (Garde [] gardes, TriValeur valeur)`

  Évalue les gardes et retourne celles évaluées à `valeur` *(*`TriValeur` *est un* `enum`*, pouvant être egal à* `vrai`*,* `faux`*, ou* `neutre`*)*

ThreadCtrl

Correspond à un vecteur d'état associé à chaque processus, et utilisé par le contrôleur. **Attributs**

- `public Semaphore sem`

  Sémaphore initialisé à 0 et permettant au processus de venir se mettre en attente.

- `public Garde [] waitingGardes`

  Liste des gardes pour lesquelles le processus est en attente. (ce qui implique que toutes ces gardes contiennent une commande d'entrée/sortie)

- `public boolean dead = true`

  Booléen indiquant si le processus est actif ou s'il a finit son exécution

Méthodes

- `public garde findMatch (Garde g)`

  Cherche dans la liste `waitingGardes` la garde pouvant correspondre à celle donnée en paramètre, ou `null` si aucune ne correspond

3 Exemple de compilation

Dans cette section va être montré un extrait du résultat de la compilation de l'exemple 1.4.1.

Les classes créées

Quatre classes ont été créées par le compilateur :

- `PRODUCTEUR.java` ;
- `CONSOMMATEUR.java` ;
- `DELEGUEUR.java` ;
- `MAIN.java`, étant l'unique classe à contenir une méthode `main`.

Exemple de compilation : la classe `CONSOMMATEUR.java`

```
public class CONSOMMATEUR extends CspThread {
   final int i;
   DELEGUEUR delegueur;

   public CONSOMMATEUR(String name, int i) {
      super (name);
      this.i = i;
   }

   public void setVariables(DELEGUEUR delegueur) {
      this.delegueur = delegueur;
   }

   public void run () {
      while (true) {
         if (debug (new DebugObj(20, 3, 20, 14))) break;
         /* nb : integer */
         final CspData<Integer> nb = CspData.newCspData(0);
         if (debug (new DebugObj(21, 3, 23, 3))) break;
         /* [ true -> ...] */
         debug (new DebugObj(22, 4, 22, 9));
         /* true */
         while (true) {
            if (debug (new DebugObj(22, 12, 22, 23))) break;
            /* delegueur?nb */
```

```
            if (!input (delegueur,new Expression("none", nb))) break;
            if (debug (new DebugObj(22, 26, 22, 35))) break;
            /* print (nb) */
            print(nb.get());
          }
          success = true;
          break;
        }
        fireDeath();
      }
}
```

4 Documents

4.1 Journal de bord

lundi 11 avril

Analyse du sujet, recherches sur CSP, latex et JAVA 1.5

-> Choix d'implémentation : java 1.5 (synchronisation plus facile)

-> Découverte de JCSP (utile ou pas ?)

mardi 12 avril

Début de prog d'un compilateur de la grammaire d'une grammaire formelle

-> but : programmer plus facilement

mercredi 13 avril

continuation de mardi

jeudi 14 avril

ajout d'un txtStack

reflexion sur l'édition des liens

vendredi 15 avril

reste qq modif à faire de la grammaire de l'analyseur,

le code postfixé ainsi que la pile des textes sont ok

-> rete à implémenter la compilation.....

-> il serait aussi préférable que la grammaire puisse intégrer un # afin

de pouvoir simplifier les appels aux fonctions de traitement de la grammaire.. a voir

lundi 18 avril

Continuation analyseur jusqu'a mardi

mercredi et jeudi 20 et 21 avril

Début controleur

vendredi 22 avril

Cahier des charges

lundi 25 avril

Cahier des charges (et réunion)

mardi 26 avril

Communication entre 2 proc OK
reste à faire la comm alternative

mercredi 27 avril

Alternative ok, mai petit prob : gestion des gardes simples (cad sans cmd d'e/s)

jeudi 28 avril

refonte du controleur
-> reste a le refaire
refonte du cdc

vendredi 29 avril

fin du cdc
Refonte du controleur..encore
Problèmes rencontrées :
les algos font du cas par cas -> c'est pas idéal
problème de performances : 1 proc a la fois dans le controleur problème : deadlock Nécessité de simplifier tout ca solutions : Trouver des points communs entre les selects et les commandes i/o pour améliorer les algos pb de performances : libérer le sémaphore principal du controleur en utilisant un sémaphore par traitement de commande

reste a faire :
refaire le controleur, et peut etre aussi les classes Select et IOCommand
Analyseur

lundi 2 mai

début refonte de la totalité du package manager (avec commentaire ce cou-ci)
Problèmes rencontrées : difficultés a utiliser java 1.5 (generics plus particulierement)
tjrs le prob du select -> niveau controleur
-> niveau écriture des progs (création répétitive du select = pas bo)

solutions
: pour le controleur -> associé a chak proc une liste de commande pour lequel il est pret, en partant du fait que cette liste ne comprendra qu'1 seul élément s'il s'agit d'une commande d'i/o std
pour le select -> peut etre englober dans le select la liste des instructions..peut etre
reste a faire : tout le reste

Mardi 3 mai

Controleur quasi fini, cas lecteurEcrivain1 OK
problèmes rencontrées : tjrs le mm probleme de surblocage -> des processus sont retardées par d'autres

mercredi 4 mai Fin de la version 0.1 du controleur (cad du zip CspProjet)
Debut d'utilisation de JFLEX
Problèmes recontrées et corriger : yen a eu... (prob d'évaluation de la garde)
Reste a faire : faire une version 0.2 du controleur inspirée de 0.1 mais plus performante (plus de sémaphore)

lundi 9 mai

Début version 0.2 du package cspManagement
-> délégation du ThreadCtrl dans chaque CspThread
-> fonction getNewSelect
-> protection des fonctions input et output
Implémentation des paradigmes : lecteur/écrivains et lecteur/ecrivains avec priorité aux écrivains

début écriture de la grammaire avec CUP -> ptits probs bizarres de conflit (ambiguité parait-il..)

Changement du "type" sous forme d'enum pour un type sous forme String (moins de vériifcation a faire)

problemes rencontrées : tjrs le pb de surblocage, plus nouveau prob : comment transmettre une variable déclarée dans
une garde a la liste de commande qui suit ?...
solutions : ...
reste a faire : analyseur

mardi 10 mai

Travail sur l'analyseur
problemes rencontrés : problemes dans la grammaire qui n'en sont pas vraiment : CUP ne semble pas terrible terrible...
solution : ba fau chercher des optimisations dans la grammaire

mercredi 11 mai

travail sur l'ananylseur

jeudi 12 mai

travail sur l'ananylseur
-> choix pour lever l'ambiguité dans la grammaire : obligation de mettre un idenfiant de processus, utilisation des crochets pour les tableaux

vendredi 13 mai

travail sur analyseur
CUP Vraiment pas bien , début d'implémentation de la grammaire avec ANTLR

lundi 16 mai

recherche sur l'analyseur -> de plus en plus difficile d'implémenter la syntaxe CSP
->solution : ANTLR propose une vérification de "prédicat" (cad qu'une suite de lexeme est verifiée entierement avant de la considérer comme bonne !)

mardi 17 mai

fin implémentation de la syntaxe, reste qq bricole (constante global, le 'moins', et dautre ptites choses)
-> début structuration programme sous forme d'objets

mercredi 18 mai

Travail sur le code intermédiare -> il sera implémenté par une structure hiérarchisée de classe JAVA.

jeudi 19 mai

rajou de qq bricoles dans la syntaxe
Début préprocessing

-> choix d'implémentation : pas de boucle FOR pour les alternatives -> préprocessing

vendredi 20 mai

travail sur l'analyseur, et sur le runtime.

problèmes rencontrés dans l'analyseur :
- portées des variables avec les commandes paralleles. Solution -> utilisation d'une méthode 'setVariable' permettant
de fournir les différentes variables nécessaires. Cependant, la compilation devient un peu plus tordue...

- utiliation des constantes -> préprocesssing

- utilisation des expression 'méga structurée' -> généralisation dans le runtime de 'Expression' et 'CspData' dans 'ExpressionValeur'

- Problème difficile : les tableaux -> nécessité d'utiliser une 3emes classe (CspArray) pour gérer le passage des tableaux lors des commande d'i/o, etc..

lundi 23 mai

Travail effectué :
- début de génération de code -> declaration et affectation OK
Problèmes rencontrés :
- chainage des listes de commande (c'est un peu l'anarchie..)
- début commande parallele -> tres tres compliquées -> nécessité de déclarer les variables externes (pour 'setVariable'), utilisation difficile des indices (passage par constructeur ou setVariable ?), création des processus fastidieuse : 3-4 boucles nécessaires ! !

mercredi 25 mai

-> ca peut marcher bientot, mais faudra refaire toute la génération de code (c'est vraiment moche ! !)

jeudi 26 mai

fin analyseur -> plein de détails sur la grammaire a faire part a O.R.

vendredi 27 mai

début interface

lundi 30 mai

interfaec quasi ok, mais prob : impossible de re-loader une classe

mardi 31 mai

resolution probleme loadation avec création de 'myLoader'
interface quasi ok encore -> ce cou-ci mm la vue des processus en cours est .. visible..
reste a faire : qq chose de plus jolie et de plus performant

mercerdi 1 juin

amélioration interface , début plan rapport

jeudi 2 juin

améliaration interface, detection et réparation erreur dans compilateur : detection de l'indicage délicat

vendredi 3 juin

interface probablement

lundi 6 juin

matinée : travail sur le rapport

aprem : interface (amélioration de la dispo des boutons) et rapport

4.2 Cahier des charges

4.3 Manuel d'utilisateur