Département
Informatique

Rapport de stage

Compilateur de programmes CSP en JAVA

Benjamin LE BOZEC

27 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

Je remercie particulièrement M. Olivier ROUX
pour m'avoir accueilli pendant dix semaines.
Je remercie aussi M. Sébastien FAUCOU
pour m'avoir aidé et conseillé tout au long du stage.

Résumé

Ce rapport s'inscrit dans le cadre d'un stage de fin d'année effectué à l'IRCCYN[1], sous la tutelle de M. Olivier ROUX. Le thème du stage était de réaliser un compilateur de programmes CSP (*Communicating Sequential Processes*, mis en place par C.A.R. HOARE en 1978) en JAVA. En d'autres termes, il fallait réaliser un outil capable de créer un programme JAVA ayant le même comportement qu'un programme CSP donné.

Dans cet optique, la première démarche fut d'étudier le langage CSP afin d'analyser ses contraintes et principes. On retiendra tout particulièrement le mécanisme de synchronisation complexe basé sur le principe du rendez-vous. Par ailleurs, la définition originale de CSP donnée par HOARE laissait quelques ambiguïtés qu'il a fallu clarifier, notamment concernant la déclaration des variables et tableaux. A l'initiative de M. Olivier ROUX, une modification notoire de la commande alternative a de plus été apportée.

Par la suite, le stage à consisté a se porter vers le langage JAVA pour en étudier les outils et les contraintes. Le choix de JAVA 1.5 comme plate-forme a par ailleurs permis une plus grande souplesse dans l'implémentation (notamment avec le principe des *génériques* ainsi qu'au travers de la nouvelle *API de concurrence*).

Parallèlement, il a aussi été nécessaire de s'intéresser aux grands principes de la compilation, depuis l'analyse lexicale jusqu'à la génération de code.

Enfin, une interface graphique a été réalisée. Ses missions étaient de faciliter l'utilisation du compilateur d'une part, et de permettre le suivi et le contrôle de l'exécution du programme CSP d'autre part. Il est important de préciser que la mise en place d'une telle interface n'a pas été simple, notamment concernant le contrôle de l'exécution qui a nécessité l'implémentation d'outils supplémentaires.

Ce stage a donc permis d'aborder de nombreux domaines, et fut particulièrement instructif, tout particulièrement en ce qui concerne le domaine de la compilation qui n'avait pas été abordé durant ma formation.

[1] IRCCYN : Institut de Recherche en Communications et Cybernétique de Nantes, URM CNRS 6597

Abstract

This report falls under the scope of a training course executed in the IRCCYN, under the supervision of Mr. Olivier ROUX. The theme was to set up a compiler of CSP programs (*Communicating Sequential Processes*, set up by C.A.R. HOARE in 1978) into JAVA programs. In other words, the purpose was to develop a tool able to create a JAVA program having the same behavior as a given CSP program.

Accordingly, the first step was to study CSP in order to analyze its constraints and principles. We will particularly remember the complex synchronization mechanism based on the *rendez-vous* principle. In addition, the original CSP definition given by HOARE involves some confusions which needed to be clarify, specially about variables and arrays declarations. With the initiative of Mr. Olivier ROUX, an important change of the alternative command was made.

Thereafter, the training consisted in focusing on the JAVA language, in order to study its tools and constraints. The choice of JAVA 1.5 as a platform can be justified by the flexibility in implementation (specially through the *generics* and the new concurrent API).

At the same time, the theory of compilation was studied, from the lexical analysis to the code generation.

At last, a graphical user interface has been set up. Its purposes was to easien the use of the compiler in one hand, and to allow the user to follow and monitor the behavior of the program throughout its run. It is important to note that developping such an interface is not easy, specially in the matter of monitoring the run which means the set up of additional tools.

Table des matières

1 Environnement

1.1 Présentation de l'institut

La présentation suivante de l'IRCCYN a été récupérée depuis leur site internet :

L'Institut de Recherche en Communications et Cybernétique de Nantes (IRCCyN - UMR CNRS 6597) est une unité mixte de recherche du Centre National de la Recherche Scientifique (CNRS), rattachée au Département Sciences et Technologies de l'Information et de la Communication (STIC), dont les tutelles sont l'École Centrale de Nantes, l'Université de Nantes et l'École des Mines de Nantes. Il relève aussi des départements Sciences Pour l'Ingénieur (SPI) et Sciences De la Vie (SDV).

Cet institut compte aujourd'hui environ 175 personnes, dont 78 chercheurs et enseignants-chercheurs, 70 doctorants et 19 ingénieurs, techniciens et administratifs.

Il est aussi soutenu par les collectivités locales et régionales du fait de sa vocation à fédérer les activités nantaises de recherche relatives à l'étude des mécanismes de communication et de contrôle dans les machines, les systèmes organisationnels et les êtres vivants (cybernétique).

La recherche à l'IRCCYN n'est pas seulement spéculative pour la production de connaissances nouvelles, mais elle est en grande partie de nature technologique, en ce sens qu'elle concerne le développement d'outils et de méthodologies capables d'apporter des solutions à des problèmes concrets issus du tissus industriel ou socio-économique.

Les recherches et les actions de valorisation qui y sont développées couvrent un domaine scientifique très large englobant :

- l'automatique ;
- le traitement du signal, des images et des signaux radar ;
- la robotique ;
- la productique ;
- la conception mécanique assistée par ordinateur ;
- les systèmes temps réels ;
- la logistique et la recherche opérationnelle ;

- la prise en compte des facteurs humains (psychologie cognitive) ;
- La qualité des réseaux.

Ceci en fait un des principaux laboratoires nationaux dans cette thématique.

1.2 Déroulement du stage

Le stage a été effectué dans l'une des salles informatique de l'IRCCYN. Un ordinateur fut mis à ma disposition, ordinateur qui a par ailleurs nécessité l'installation de la dernière version de JAVA (1.5) et du logiciel ECLIPSE.

2 Expression de la demande et première approche du sujet

CSP (*Communication Sequential Processes*) est un langage de programmation parallèle(ou concurrente). Il repose sur quelques principes algorithmiques simples (alternative et répétition essentiellement) et propose un mécanisme de synchronisation de processus fondé sur le principe du rendez-vous.

L'objectif du travail est de mettre au point un compilateur qui transforme des programmes CSP en JAVA, dans le but de d'obtenir un programme JAVA **ayant le même comportement** que le programme CSP source. Le choix de JAVA comme langage destination peut être justifié par la grande palette d'outils qu'il fournit d'une part(la dernière version de JAVA apporte en effet de nombreux mécanismes de synchronisation), et d'autre part par le fait qu'il soit compatible sur les différents systèmes existants (Microsoft Windows, Linux, ...). C'est d'ailleurs pour ce second atout que le compilateur sera lui-même implémenté en JAVA.

Dans un premier temps, il convient donc de comprendre le langage CSP et d'analyser ses contraintes. Parallèlement, il est nécessaire de rechercher quels outils fournis par JAVA peuvent être utilisés pour implémenter les mécanismes de CSP. Ce travail de recherche et d'analyse devra donc aboutir à l'élaboration d'outils visant à simplifier la compilation, tel qu'un contrôleur pour gérer les communications inter-processus.

Dans un second temps, le travail se portera alors sur le processus de compilation, c'est-à-dire sur la traduction propre des programmes CSP en programmes JAVA.

Par ailleurs, une interface graphique a été réalisée afin de faciliter l'utilisation du compilateur. Cet interface a aussi pour vocation de permettre à l'utilisateur de suivre l'exécution du programme. Un manuel d'utilisateur est fourni en annexe pour expliciter le fonctionnement de cette interface.

Architecture de l'application

On peut donc considérer que l'application à réaliser sera composée de trois modules :

- un module **compilateur**, centre de l'application, chargé de transformer un programme CSP en programme JAVA ;
- un module **interface** ;
- un module **runtime**, englobant les outils nécessaire à l'exécution du programme JAVA résultant de la compilation.

3 Le langage CSP

CSP a été mis au point par HOARE dans un article publié en 1978[2]. Comme expliqué précédemment, c'est un langage de programmation parallèle qui intègre un mécanisme de synchronisation basé sur le principe du rendez-vous (détaillé plus loin au travers de la commande d'entrée/sortie). Combinant ce mécanisme à une syntaxe simple et concise, CSP permet alors l'implémentation rapide des paradigmes classiques de la concurrence, tels que producteurs/consommateurs ou lecteurs/écrivains.

Un exemple de programme CSP tel que le compilateur doit l'accepter (c'est-à-dire tenant compte des extensions définies dans la partie 3.3) et détaillé est disponible en annexe.

D'autres exemples non commentés sont par ailleurs disponibles, implémentant des paradigmes classiques de lecteurs/écrivains avec priorité aux écrivain, producteurs/consommateurs avec tampon, etc.

3.1 Généralités

Un programme CSP se présente sous la forme d'une suite de commandes et de déclarations de variable séparées par des points-virgule. HOARE distingue alors deux types de commandes :

- les commandes simples : commande nulle (`skip`), commande d'affectation et commande d'entrée/sortie ;
- les commandes structurées : commande parallèle, commande répétitive et commande alternative.

Les commandes d'entrée/sortie, parallèle, répétitive et alternative seront détaillées par la suite. Les autres commandes ne nécessitant pas d'explications particulières, elles sont simplement reprises en annexe.

Notons aussi que l'échec d'une commande entraîne l'échec du processus ou de la commande structurée qui la contient. Sauf dans le cas particulier d'une répétitive, dont l'échec d'une commande interne entraîne sa terminaison et non sa faillite (cf. partie 3.2.4, **La commande répétitive**).

Une notion de signal a aussi été introduite en CSP. Un signal est en quelque sorte une variable complexe (ou structurée) composé d'un constructeur (un identifiant libre) ainsi que d'un ensemble de valeurs. Les valeurs peuvent être soit de type élémentaire (entier, chaîne de caractères, etc.) soit

[2]Communicating Sequential Processes, C.A.R HOARE, 1978 : www.cs.virginia.edu/ evans/crab/hoare1978csp.pdf

d'autres signaux.

Exemples :

- `p(3,5)` (le constructeur est `p` et contient les valeurs `3` et `5`)
- `p(q(3), 5)`
- `p()` (ici, on dit que `p()` est un signal pur puisqu'il ne contient aucune valeur)

Notons aussi qu'un signal peut être affecté à une variable (exemple : `x := p (4)`).

Enfin, du point de vue de la portée des variables, les variables définies avant une commande structurée doivent être visibles par les commandes qu'elle contient.

3.2 Détails sur les commandes

3.2.1 La commande parallèle

Elle permet la mise en concurrence de processus par la syntaxe suivante :

```
[
   proc1 :: liste de commandes
||
   proc2 :: liste de commandes
||
   ...
]
```

Une telle commande ne se termine que lorsque tous les processus qu'elle définit sont terminés, l'échec d'un seul entraînant l'échec de la commande parallèle.

Rappelons ici qu'une variable définie avant une commande structurée doit être visible par les commandes qu'elle contient. Dans le cas d'une commande parallèle, ceci implique nécessairement que tous les processus définis devront avoir accès aux variables déclarées avant la commande parallèle. On assiste donc ici à un **partage de ressources**, dont les contraintes soulevées (accès exclusive, etc.) sont à la charge du programmeur.

On note aussi la possibilité de déclarer plusieurs processus pour une même liste de commandes, comme le montre l'exemple suivant :

```
proc1 [i :1..10] : : liste de commandes
```

Ici, dix processus ayant la même liste de commandes seront mis en concurrence.

3.2.2 La commande d'entrée/sortie

Les communications entre processus reposent sur cette commande. Pour expliciter son fonctionnement, partons du cas simple de deux processus `producteur` et `consommateur`, le premier voulant transmettre une variable `x` au second. Une telle communication s'effectuera alors au travers des commandes suivantes :

- `consommateur !x` permettra au processus `producteur` d'envoyer `x` au processus `consommateur`
- `producteur ?x` permettra au processus `consommateur` de recevoir `x` depuis le processus `producteur`

Du point de vue de la syntaxe, on remarque que le `!` indique une commande de sortie, tandis que le `?` indique une commande d'entrée. Plus formellement, une commande d'entrée se présentera sous la forme :

```
processus_source ?variable_cible
```

et une commande de sortie sous la forme :

```
processus_cible !expression_source.
```

De plus, pour que la communication soit possible entre deux commandes d'e/s, CSP impose que :

1. l'une soit une commande d'entrée, l'autre une commande de sortie ;
2. les deux processus se nomment mutuellement ;
3. les typages de la variable cible et de l'expression source soient identiques.

Dans ces conditions, la transmission peut avoir lieu, et le contenu de l'expression source est copié vers la variable cible. Si l'une de ces conditions n'est pas respectée, les processus sont mis en attente, entraînant par conséquent un interblocage. Si l'un des processus est *mort*, alors toute commande d'entrée/sortie impliquant ce processus doit échouée. Enfin, le premier processus demandant la communication doit être mis en attente jusqu'à ce que le second le rejoigne. On retrouve ici *le principe du rendez-vous.*

Notons aussi le cas particulier d'une synchronisation simple, c'est-à-dire sans transmission de valeur, possible grâce à l'utilisation d'un signal pur.

Exemple :

```
[
   producteur :: ... consommateur?p(); ...
||
   consommateur :: ... producteur!p(); ...
]
```

3.2.3 La commande alternative

Une commande alternative se présente sous la forme d'un ensemble de sélective, chacune étant composée d'une garde ainsi que d'une liste de commandes. Une garde est quant à elle composée d'une partie `expression booléenne` et d'une partie `commande d'entrée`, l'une ou l'autre pouvant être omise.

Syntaxiquement, une commande alternative se présente sous la forme suivante :

```
[
   garde1 -> liste de commandes
[]
   garde2 -> liste de commandes
[]
   ...
]
```

Lors de l'exécution d'une commande alternative, chacune de ses gardes est testée, afin de déterminer sa valeur selon une logique *trivaluée* (c'est-à-dire qu'une garde peut être **vrai**, **fausse**, ou **neutre**) :

- Une `expression booléenne` peut évidemment prendre les valeurs **vrai** ou **faux**
- Une `commande d' entrée` prend la valeur :
  - **vrai** si la communication est possible **immédiatement** ;
  - **neutre** si le processus nommé par la commande n'est pas encore prêt ;
  - **faux** si le processus nommé par la commande est mort (i.e. a fini son exécution).

Dans le cas d'une garde contenant à la fois une partie `expression booléenne` et une partie `commande_entrée`, l'expression booléenne est d'abord évaluée. Si elle est à **faux**, la garde est évaluée

à **faux**. Si elle est à **vrai**, la commande d'entrée est alors évaluée, donnant ainsi la valeur à la garde.

Ainsi, si une ou plusieurs commandes gardées sont **vraies**, un choix **indéterministe** (i.e. aléatoire) doit être effectué pour n'en sélectionner qu'une. Si aucune n'est **vraie** mais que certaines sont **neutres**, le processus se met en attente des commandes d'entrées correspondantes. Et si toutes les commandes gardées sont **fausses**, la commande alternative échoue.

Si une garde est sélectionnée, la liste de commande correspondante doit alors être exécutée.

Il est aussi important de noter que la contrainte d'ordre syntaxique limitant les commandes d'e/s dans les gardes aux simples commandes d'entrée, provient d'un choix fait par HOARE dans le but d'éviter les incohérences.

Pour illustrer ce propos, partons de l'exemple suivant (qui suppose possible l'utilisation de commande de sortie dans les gardes) :

```
[
   proc1 ::
         [
           proc2?p() -> ...
         []
           proc2?q() -> ...
         ]
||
   proc2 ::
         [
           proc1!p() -> ...
         []
           proc1!q() -> ...
         ]
]
```

Supposons que les deux processus soient chacun *arrivés* sur leur commande alternative. Les deux commandes vont donc être évaluées **parallèlement** :

- le premier processus va évaluer chacune des gardes de l'alternative, puis, les deux étant `vrai`, il va effectué un choix aléatoire entre l'une ou l'autre ;
- de même, le second processus va se retrouver à devoir faire un choix entre l'une ou l'autre des gardes de son alternative.

Si les deux alternatives sélectionnent la même communication, aucun problème n'est soulevé. Par contre, si chacune choisie une communication différente, on assiste à un cas d'incohérence entraînant nécessairement un interblocage.

Ainsi, la communication entre commandes d'entrée/sortie gardées posant des problèmes, HOARE a décidé de l'empêcher en autorisant uniquement les commandes d'entrée dans les gardes.

Notons que cette contrainte très restrictive à fait l'objet d'une modification. (cf paragraphe 3.3.3).

3.2.4 La commande répétitive

La commande répétitive est composée d'une unique commande alternative, dont l'échec entraîne la fin de la répétition. On peut donc considérer qu'une répétitive ne peut que se terminer qu'avec succès.

Syntaxiquement, elle se présente sous la forme d'une étoile suivie d'une alternative :

```
*commande alternative
```

Par exemple :

```
i := 0;
*[
     i < 10 -> i := i + 1
]
```

Ici, la commande répétitive se terminera lorsque `i` aura atteint la valeur `10`.

3.3 Extensions faites au langage

Voici la liste des extensions faites à CSP :

1. la syntaxe des déclarations des variables et tableaux a été explicitée ;
2. la possibilité de déclarer des processus en dehors d'une commande parallèle a été rajoutée ;
3. la déclaration d'un processus `MAIN` a été imposée afin d'indiquer quel sera le premier processus à exécuter ;
4. la possibilité de déclarer des constantes à aussi été rajoutée ;
5. deux nouvelles commandes ont été définies (`print` et `sleep`) ;
6. et enfin une modification notoire de la commande alternative a été faite.

Les points 1, 2 et 4 sont principalement des apports syntaxiques et sont repris en annexe. Le troisième point ne nécessite pas d'explication particulière (pour une clarification éventuelle, se référer aux exemples fournis en annexe). Les autres extensions sont détaillées dans la suite du rapport.

3.3.1 La commande `print`

Dans les exemples donnés dans l'article original de CSP, HOARE utilisait un processus `console` afin d'afficher à l'écran les différents messages que pouvaient produire les processus. Ainsi, pour afficher le contenu d'une variable `message`, la commande pouvait ressembler à ceci :

```
console !message
```

Le processus `console` était donc un processus toujours présent (c'est-à-dire automatiquement lancer a l'exécution de tout programme CSP), et pouvant communiquer avec **tous** les processus (sans avoir à les citer explicitement dans une commande d'entrée/sortie, comme l'exige cependant CSP).

Pour palier aux problèmes que pouvait soulever l'intégration d'un tel processus, la création d'une commande `print` a été préférée. Cette commande permet à chaque thread d'afficher un message sur la sortie standard, prenant en entrée n'importe quelle expression :

```
print(expression)
```

3.3.2 La commande `sleep`

Cette commande a simplement été rajoutée afin de permettre l'interruption momentanée d'un processus. Elle se présente sous la forme suivante :

```
sleep (temps)
```

`temps` étant exprimé en millisecondes.

L'utilité d'une telle commande sera, par exemple, de simuler un accès long à une ressource, ou même de permettre l'implémentation de processus `horloge`, `chien de garde`, etc.

Un exemple concret d'utilisation est donné en annexe au travers du cas faisant intervenir un chien de garde (annexe 1.4.4).

3.3.3 Modification de la commande alternative

Une modification assez importante de la commande alternative a été faite. Comme expliqué précédemment, une commande alternative est composée d'une liste de gardes, ces gardes pouvant contenir une commande d'entrée mais **pas de commande de sortie**. Cette contrainte d'ordre syntaxique (justifiée dans le paragraphe 3.2.3) a été levée afin de permettre plus de souplesse dans le programme. Á la place, une condition supplémentaire a été rajoutée aux conditions d'établissement d'une communication entre deux processus. Maintenant, pour qu'une transmission soit possible entre deux commandes d'e/s, il faut que les conditions suivantes soient remplies :

1. l'une soit une commande d'entrée, l'autre une commande de sortie ;
2. les deux processus se nomment mutuellement ;
3. les typages de la variable cible et de l'expression soient identiques ;
4. **l'une des deux commandes d'entrée/sortie doit être hors garde**.

Ainsi, on évite bien les incohérences pouvant résulter d'une communication entre deux gardes de deux commandes alternatives.

4 Implémentation de CSP en JAVA

Dans cette partie vont être présentés :

1. Les différents outils fournis par JAVA et pouvant être utilisés pour implémenter les mécanismes de CSP ;
2. Les différents outils créés (tel que le contrôleur) ;
3. Un exemple de compilation.

4.1 Les outils fournis par JAVA

4.1.1 L'API de concurrence

JAVA 1.5 a enrichi l'API classique d'un nouveau package *java.util.concurrent*, contenant de nombreux outils de synchronisation, tels que les loquets, les barrières, les sémaphores ou même les échangeurs, chacun étant détaillé plus loin. Cependant, seuls les sémaphores ont été utilisés, les autres outils n'offrant que peu d'intérêt ici.

Les loquets permettent de bloquer un processus jusqu'à ce qu'un certain nombre de tâches aient été effectuées. Un tel mécanisme n'a pas vraiment d'utilité ici.

Les barrières correspondent à un rendez-vous à plusieurs threads. En effet, chaque thread a la possibilité de venir se bloquer sur une barrière jusqu'à ce que tous les processus attendus soient eux aussi venus se bloquer. Tous sont alors libérés. Un tel outil aurait pu s'avérer utile, mais l'utilisation des sémaphores a été favorisée, ces derniers fournissant une utilisation plus simple et moins propice aux erreurs.

Les échangeurs sont très proches de la synchronisation CSP. En effet, cet outil permet à deux processus de s'échanger des informations de même type, mettant en attente le premier processus arrivé. Cependant, la commande alternative aurait été difficilement implémentable, cet outil n'offrant que peu de souplesse d'une part, et ne correspondant pas tout à fait à la sémantique de CSP d'autre part. En effet, la transmission d'information est réalisée dans les deux sens, tandis que CSP ne permet que des transmission unilatéral.

Les sémaphores ont eux aussi été implémentés dans ce nouveau package JAVA. Rappelons qu'un sémaphore a comme principale utilité d'assurer une forme d'exclusion mutuelle, mais il peut aussi

permettre de bloquer un processus jusqu'à ce qu'un autre décide de le libérer (cas où le sémaphore est initialisé avec 0 *jetons*). C'est donc surtout cette deuxième fonction qui nous intéressera ici.

Les verrous existaient déjà avant la version 1.5. Ils correspondent à un mécanisme relativement complexe, permettant en autre d'assurer à un processus la manipulation exclusive d'un objet. Ici, les sémaphores ont été préférés à ce mécanisme.

4.1.2 La création de processus en JAVA

En JAVA, tout processus doit hériter de l'interface `Runnable`, obligeant ainsi l'implémentation de la méthode `run` qui est exécutée au lancement du processus. La classe `Thread` permet alors la manipulation de ces processus, se chargeant entre autre de les exécuter.

Notons aussi la méthode statique `sleep` de la classe `Thread`, qui pourra être utilisée pour la compilation de la commande `sleep`.

4.2 Les outils créés : package *csp_ runtime*

Le package *csp_ runtime* contient un ensemble de classes et d'interfaces visant à simplifier l'exécution des programmes CSP. On peut citer :

- la classe `CspThread` qui correspond à la classe parente de tous les processus CSP ;
- la classe `CspData` qui est la classe parente de toutes les variables ;
- la classe `CspArray` permettant de définir des tableaux ;
- la classe `Signal`, correspondant à la notion de signal CSP ;
- la classe `Select`, permettant de simplifier les commandes alternatives ;
- la classe `Garde` simplifiant la manipulation des gardes des commandes alternatives ;
- la classe `CommandeES` correspondant à une commande d'entrée/sortie CSP ;
- l'interface `ExprBool` qui permet la déclaration de la partie booléenne d'une garde ;
- l'interface `Valeur`, qui est implémenté par toutes les classes stockant des données (c'est-à-dire les classes `CspData`, `CspArray`, et `Signal`) ;
- la classe `Controleur` venant gérer les communications inter-processus ;
- et enfin la classe `ThreadCtrl`, correspondant à un vecteur d'état associé à chaque processus, et utilisé exclusivement par le contrôleur.

En annexe sont fournies des précisions sur chacune de ces classes et interfaces à la manière d'une documentation JAVA.

4.2.1 Implémentation des processus : classe `CspThread`

Pour chaque processus défini dans un programme CSP, une nouvelle classe héritant de `CspThread` devra être créée. `CspThread` fournit par ailleurs quelques méthodes afin de simplifier la compilation :

- La méthode `start` permettant de lancer le processus ;
- La méthode `join` permettant d'attendre la fin de l'exécution d'un processus ;
- Les méthodes `input` et `output`, correspondant respectivement à une commande d'entrée et une commande de sortie ;
- enfin la commande `fireDeath` permettant au processus d'indiquer la fin de son exécution.

4.2.2 Implémentation des variables : la classe `CspData`

Comme cela va être expliqué par la suite, un contrôleur sera chargé de gérer les commandes d'e/s. Par conséquent c'est à lui que revient la charge d'effectuer la transmission et donc d'affecter la valeur de l'expression source à la variable cible (cf commande entrée/sortie). Pour que cette trans-

mission se fasse correctement, il est donc nécessaire au processus émettant la commande d'entrée de fournir une **référence** à la variable cible.

En JAVA, il n'est pas possible de passer directement un pointeur vers une variable, et il a donc fallu créer une nouvelle classe englobant toutes variables : c'est la classe `CspData`.

Du point de vue structurel, elle contient un paramètre `value` contenant la valeur de la variable, accessible au travers de méthodes `get` et `set`. Une méthode `matches` a aussi été déclarée afin de vérifier si deux variables sont de même type (utile lors de la vérification des contraintes d'établissement de communication par le contrôleur).

Il est aussi important de noter que la classe `CspData` est de type **générique** (cette notion ayant été apportée avec JAVA 1.5), permettant une manipulation plus facile de la valeur de la variable.

Par exemple, une variable déclarée en CSP par :

```
x : integer := 0
```

sera déclarée en JAVA par :

```
CspData<Integer> x = new CspData<Integer>(0)
```

4.2.3 Implémentation des tableaux : classe `CspArray`

En CSP, les tableaux peuvent avoir plusieurs dimensions, chacune ayant une borne inférieure ainsi qu'une borne supérieure. JAVA lui n'a pas cette notion de bornes inférieure et supérieure. Ainsi, afin de simplifier la compilation, une nouvelle classe, la classe `CspArray` a été créée. Elle peut être instanciée pour un type donnée (utilisation des **génériques**), les dimensions pouvant être bornées *à la CSP*.

Une méthode `matches` a là aussi été implémentée afin de tester la correspondance de type entre deux tableaux.

Les instructions :

```
tab : [1..10] integer := [1..10] 0 ;
tab[1] := 5
```

seront donc compilées par :

```
CspArray<Integer> tab = new CspArray<Integer>(0, new Borne(1,10));
tab.set(1,5);
```

Notons par ailleurs l'utilisation de la classe `Borne`, englobant les deux bornes d'une dimension.

4.2.4 Implémentation des signaux : classe `Signal`

Comme expliqué dans la partie 3.1 , un signal est caractérisé par un constructeur ainsi qu'un ensemble de valeurs. Ici, le constructeur sera implémenté par une chaîne de caractères, les valeurs étant de type `Valeur`, et pouvant donc être des objets de type `CspData`, `CspArray` ou `Signal`. Ici, pas de méthode `get` ou `set`, mais une méthode `copyFrom`, permettant de récupérer les valeurs d'un autre signal de même constructeur, et de les affecter aux valeurs de l'objet courant.

Par exemple, les instructions :

```
x : p(integer, integer) := p(1,2);
x := p (3,5)
```

seront compilées par :

```
Signal x = new Signal ("p", new CspData<Integer>(1), new CspData<Integer>(2));
x.copyFrom (new Signal("p", new CspData<Integer>(3), new CspData<Integer>(5)));
```

Même si une telle compilation peut paraître lourde, elle n'en reste pas moins puissante et permet, entre autre, des affectations du type :

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

En effet, la méthode `copyFrom` utilise un tampon afin de palier aux problèmes qui peuvent être posés par de telles instructions.

4.2.5 Gestion des communications : le contrôleur

Le contrôleur est l'outil permettant de gérer les synchronisations telles que CSP les définit, en jouant le rôle d'intermédiaire pour chaque communication. Ainsi, les commandes d'entrée/sortie ne s'adresseront pas directement au processus avec lequel elle doivent communiquer, mais au contrôleur qui se chargera alors de vérifier les différentes contraintes imposées par CSP, puis d'effectuer la

transmission.

De même, le contrôleur est chargé de choisir la sélective à exécuter dans le cas d'une commande alternative (les gardes des sélectives pouvant en effet contenir une commande d'e/s).

Ainsi, cet outil devra fournir deux accès (concrètement deux méthodes) :

- un premier accès pour la gestion d'une commande d'e/s simple (c'est-à-dire hors garde) qui prendra donc en entrée un objet `CommandeES` et fournira en sortie `vrai` si la commande a réussi ou `faux` si elle a échoué :

  ```
  gestionCommandeES (cmd : CommandeIO) retourne booléen
  ```

- un second accès pour les commandes alternatives, prenant en entrée une liste de gardes (objet `Garde`) et indiquant en sortie la garde sélectionnée ou `null` si aucune garde n'est valide (si toutes les gardes sont `fausses`) :

  ```
  gestionAlternative (gardes : Garde []) retourne Garde
  ```

Cependant, un tel schéma impliquerait la construction de deux algorithmes. Rappelons alors qu'une garde est composée d'une expression booléenne et d'une commande d'e/s, l'une ou l'autre de ces parties pouvant être omise. Une simplification possible pourrait donc être de considérer une commande d'e/s comme une garde sans partie booléenne. Le premier accès deviendrait donc inutile et le contrôleur serait simplifié à une seule méthode traitant une liste de gardes, cette liste ne contenant qu'un seul élément dans le cas d'une commande d'e/s simple.

Du point de vue du comportement du contrôleur ceci ne poserait aucun problème :

- Dans le cas d'une garde unique ne contenant qu'une commande d'e/s, par exemple :

  ```
  [ delegueur?msg -> ... ]
  ```

  le contrôleur doit exécuter la commande si elle est évaluée à `vrai` (ici, si le processus `delegueur` est déjà en attente), il doit placer le processus en attente si elle est évaluée à `neutre` (si `delegueur` n'est pas encore prêt), et enfin il doit indiquer que la commande a échoué si elle est évaluée à `faux` (si `delegueur` a déjà fini son exécution) ;
- Dans le cas d'une commande d'e/s hors garde, le contrôleur doit l'exécuter si elle est évaluée à `vrai`, mettre en attente le processus si elle est évaluée à `neutre`, ou indiquer que la commande a échoué si elle est évaluée à `faux`.

Les deux comportements sont bien identiques, et le contrôleur pourra donc les gérer de la même manière.

Cependant, il est important de souligner qu'une telle assimilation permettra possible la communication de deux commandes d'entrée/sortie **gardées** (à la condition qu'au moins une des deux soit **l'unique** garde de l'alternative) ce qui ne respecterait plus la dernière condition d'établissement d'une communication décrite dans le paragraphe 3.3.3. Rappelons alors que le problème que posait la communication inter-gardes résultait principalement du fait que l'une et l'autre des commandes alternatives concernées contenaient plus d'une sélective. Dans le cas d'une alternative à une sélective unique, le problème n'apparaît pas.

L'assimilation peut donc être validée.

Voici donc l'algorithme qu'implémente la méthode principale `gestionGardes`, correspondant à la méthode `gestionAlternative` citée ci-dessus.

Il prend en entrée :
- le processus `source` émetteur de la demande de communication ;
- une liste d'objets `Garde` ;

et fournit *indirectement* en sortie la garde sélectionnée, en mettant son attribut `choosen` à `vrai`. Toutes les gardes fournies en entrée doivent donc avoir leur paramètre `choosen` initialisé à `faux`.

Rappelons que les classes `Garde`, `CspThread`, `ThreadCtrl` et `CommandeES` utilisées dans l'algorithme sont détaillées en annexe (section 2).

```
Semaphore mutex = new Semaphore(1);

méthode gestionGardes (source : CspThread; gardes : Garde [])
début
      mutex.P ();

      /* filtre de la liste de garde : on ne conserve que celles évaluées à VRAI */
      trueGardes = filtreGardes (gardes, VRAI);

      si (trueGardes.taille == 0) alors
         /* aucune garde n'est directement sélectionnable
         on récupère alors les gardes NEUTRES */
         neutreGardes = filtreGardes (gardes, NEUTRE);

         si (neutreGardes.taille == 0) alors
               /* toutes les gardes sont FAUSSES, on relâche alors le sémaphore principal
               et on sort de l'algorithme */
               mutex.V ();

         sinon
               /* Sinon on se met en attente des gardes neutres (plus précisément des
               commande d'e/s qu'elles peuvent contenir) après avoir relaché le
               sémaphore principal */
               source.ctrl.waitingGardes := neutreGardes;
               mutex.V ();
               source.ctrl.sem.P ();

         fsi;
      sinon
         /* si des gardes sont exécutables, on en prend une au hasard */
          Garde garde := trueGardes[random (1, trueGardes.taille)];

         /* on indique que la garde a été choisie */
         garde.choosen = true;

         si (garde.cmdES != null) alors
               /* s'il existe une commande d'e/s, on commence par récuperer le
```

```
            processus destination */
            CspThread dest := garde.cmdES.dest;

            /* puis récupere la "garde complémentaire" de la commande choisie,
            et on effectue la transmission. */
            Garde otherGarde := dest.ctrl.findMatches (garde);
            garde.cmdES.perform (otherGarde.cmdES);

            /* enfin, on indique à l'autre garde qu'elle a été choisie,
            puis on libère le processus distant après lui avoir retirer
            toutes ses gardes en attente */
            dest.ctrl.waitingGardes := [];
            otherGarde.choosen := true;
            dest.ctrl.sem.V ();
        fsi;

        /* finalement, on libère le sémaphore principale */
        mutex.V ();
    fsi;
fin
```

Commentaires sur l'algorithme

Dans cet algorithme, on remarque donc que c'est la méthode `filtreGardes` qui se charge de vérifier les quatre contraintes à respecter pour l'établissement d'une communication (cf. partie 3.3.3), puisque c'est elle qui est chargée d'évaluer les gardes et donc les éventuelles commandes d'entrée/-sortie .

Par ailleurs, on peut remarquer la présence d'un sémaphore `mutex`, permettant de ne traiter qu'une demande de communication à la fois.

Pour conclure, voici un chronogramme pouvant illustrer le fonctionnement de l'établissement d'une communication entre deux processus `proc1` et `proc2` :

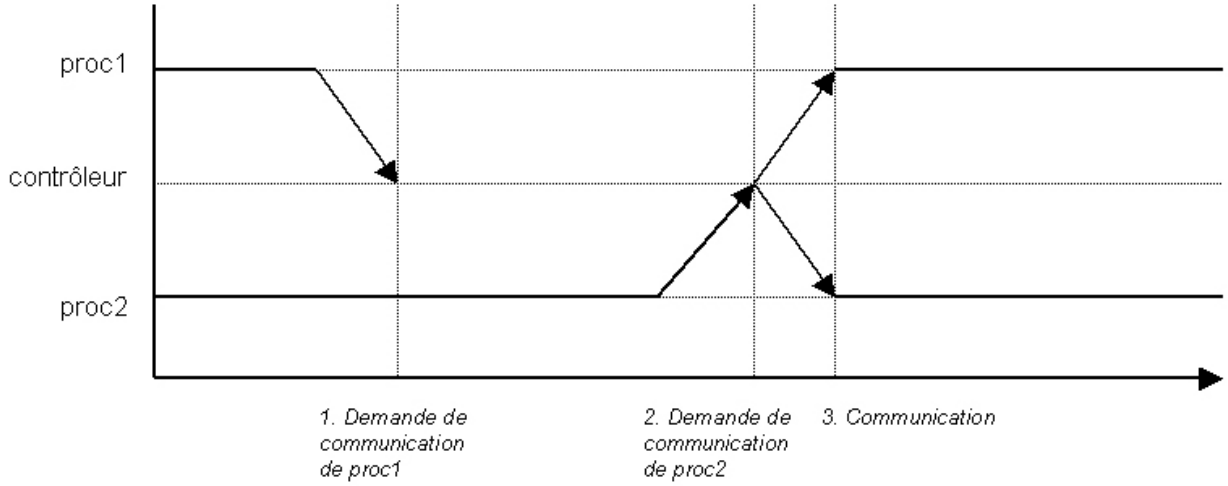

FIG. 1 – **Chronogramme d'établissement d'une communication**. Les traits pleins correspondent à l'exécution d'un processus. On remarque donc que le contrôleur stoppe l'exécution du premier processus demandant la communication, puis la rétablit une fois que la communication a été effectuée.

4.3 Exemple de compilation

Un exemple de compilation est fourni en annexe afin d'expliciter au mieux la correspondance JAVA des commandes CSP.

5 Le processus de compilation

La compilation est un procédé bien connu en Informatique, qui suit, comme c'est le cas ici, le schéma suivant :

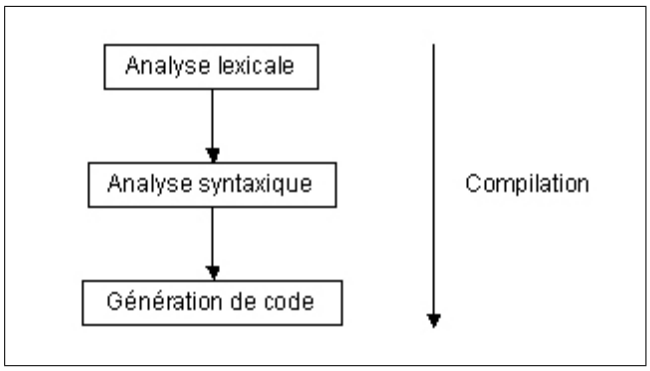

FIG. 2 – Déroulement d'une compilation

5.1 Analyse lexicale et analyse syntaxique

L'analyse lexicale consiste à découper une suite de caractères donnée en entrée (comme par exemple un fichier) en une suite de lexèmes (aussi appelés symboles), d'après des règles de substitutions bien précises, souvent données sous la forme d'expressions régulières. Exemple, un nombre entier peut être représenté par la règle :

```
nombre == [0-9] [0-9]*.
```

L'analyseur syntaxique prend quant à lui la suite de lexèmes générée par un analyseur lexical, puis cherche à quelles règles grammaticales cette suite peut correspondre.

L'exemple suivant explicite ce fonctionnement :

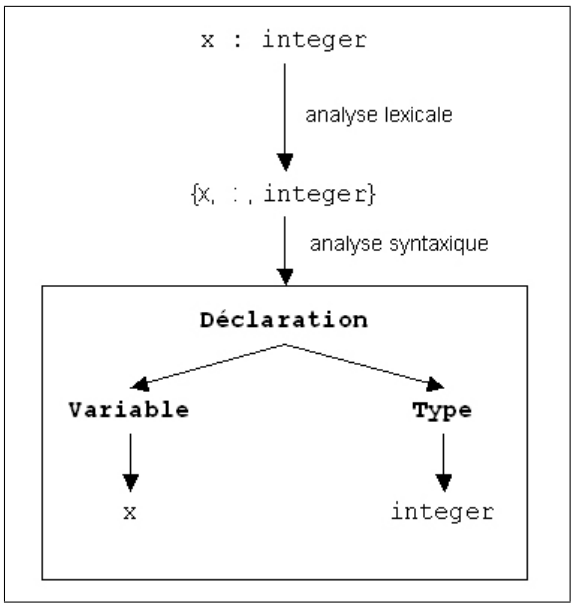

FIG. 3 – Exemple d'analyse lexicale et syntaxique

Afin d'implémenter de tels analyseurs, plusieurs outils existent déjà en JAVA :

- **JFlex**[3] : créé des analyseurs lexicaux à partir de règles semblables à des expressions régulières ;
- **CUP**[4] : fonctionne avec JFlex et créé des analyseurs syntaxiques à partir de règles grammaticales ;
- **ANTLR**[5] : créé à la fois des analyseurs lexicaux et syntaxiques.

5.1.1 Comparaison JFLex-CUP / ANLTR

Dans un premier temps, l'utilisation de JFLex couplée avec CUP avait été préférée, ces deux outils étant nettement plus simples à utiliser que ANTLR. Cependant, des problèmes d'ambiguïté dans la syntaxe ont finalement forcé l'utilisation de cet outil, difficile à prendre en main mais très puissant.

En effet, CUP est un analyseur de la famille LALR(k)[6], plus précisément un LALR(1). Il permet

[3] JFlex, developpé par Elliot BERK : http ://jflex.de

[4] CUP, developpé par Scott E. HUDSON : http ://www.cs.princeton.edu/ appel/modern/java/CUP/

[5] ANTLR : http ://www.antlr.org

[6] LALR(k) : Look Ahead Left to right with Rightmost derivations, signifiant analyse ascendante en fonction des k prochains lexèmes

donc la réalisation d'analyseurs simples et rapides, mais se retrouve très contraignant sur le type de syntaxe acceptée.

Pour illustrer ceci, partons des exemples de règles suivants issus de la grammaire de CSP (rappelons que le formalisme utilisé pour l'expression des règles syntaxiques est explicité en annexe 1.5.1) :

```
commandeES ::= commandeEntree | commandeSortie
commandeEntree ::= nomProcessus (indicage)? "?" variableCible
commandeSOrtie ::= nomProcessus (indicage)? "!" expressionSource
indicage ::= "[" indice ("," indice)* "]"
indice ::= variable | nombre
```

Précisons que ces règles permettent l'écriture de commandes telles que : `lecteur[1] !message` ou `ecrivain ?message`, etc.

On remarque ici que la différence entre une commande d'entrée et une commande de sortie ne peut être détectée que lorsque l'analyseur rencontre l'un ou l'autre des symboles `?` ou `!`. Par exemple, pour l'entrée suivante(extraite de la commande `lecteur[1] !message`) :

```
{"lecteur", "["}
```

il n'est pas possible à un analyseur LALR(1) de détecter s'il s'agit d'une commande d'entrée ou de sortie. D'ailleurs, aucun analyseur LALR(k) ne peut la détecter : le nombre d'indices peut varier de 0 à l'infini (théoriquement évidemment), et par conséquent il n'est pas possible de trouver un entier `k` **fixe** suffisamment grand pour lever l'ambiguïté.

ANTLR (qui est de type LALL(k)[7], avec k paramétrable) propose quant à lui des mécanismes permettant de lever ce type d'ambigüité. En effet, il lui est possible de vérifier si une règle est applicable **avant** de l'appliquer (c'est ce que ANLTR appelle *la prédiction syntaxique*).

Par conséquent, la règle

```
commandeES : := commandeEntree | commandeSortie
```

est décrite avec ANTLR par :

[7] LALL(k) : Look Ahead Left to right with Leftmost derivations, signifiant analyse ascendante en fonction des k prochaines lexèmes

```
commandeES :
   (commandeEntree) => commandeEntree
   |
   (commandeSortie) => commandeSortie
```

La syntaxe `(a) => a` indiquant que `a` doit être vérifié avant d'être appliqué.

C'est donc principalement pour cette raison que ANTLR a été préférée à CUP pour implémenter la grammaire de CSP.

5.2 Utilisation de ANTLR

Pour créer un analyseur syntaxique, ANTLR compile chaque règle grammaticale sous la forme d'une méthode. La règle décrite dans la section précédente sera par exemple compilée par la méthode suivante :

```
public void commandeES () {
   ...
}
```

Ainsi, la syntaxe CSP contenant une règle `programme`, racine de toute la grammaire, la vérification syntaxique d'un programme CSP s'effectuera en appelant la méthode `programme` de l'analyseur créé par ANTLR. Tout erreur dans la syntaxe étant alors indiquée par la levée d'une exception.

Par ailleurs, ANTLR apporte aussi la possibilité d'intégrer dans l'analyseur qui va être créé des instructions JAVA *libres*, permettant la construction de toute une structure de classes en même temps que se déroule l'analyse syntaxique.

Pour illustrer ce propos, l'exemple décris dans la section précédente pourra être transformer de la manière suivante : **Attention** : *Il est important ici de bien noté que la classe `CommandeES` utilisée ici ne correspond en aucun cas à la classe `CommandeES` du package `csp_runtime`.*

```
commandeES returns [CommandeES cmd]:
   (commandeEntree) => cmd=commandeEntree
   |
   (commandeSortie) => cmd=commandeSortie
```

et sera compilé par ANTLR de la manière suivante :

```
public CommandeES commandeES () {
   CommandeES cmd;
   ...
      cmd = commandeEntree ();
   ...
      cmd = commandeSortie ();
   ...
   return cmd;
}
```

Ici, il a donc été décidé d'utiliser au maximum les possibilités fournies par ce compilateur d'analyseur, dans le but d'obtenir, en fin d'analyse, un unique objet `Programme`, contenant de manière hiérarchisée l'ensemble du programme sous forme d'objets JAVA, comme l'explicite le schéma suivant :

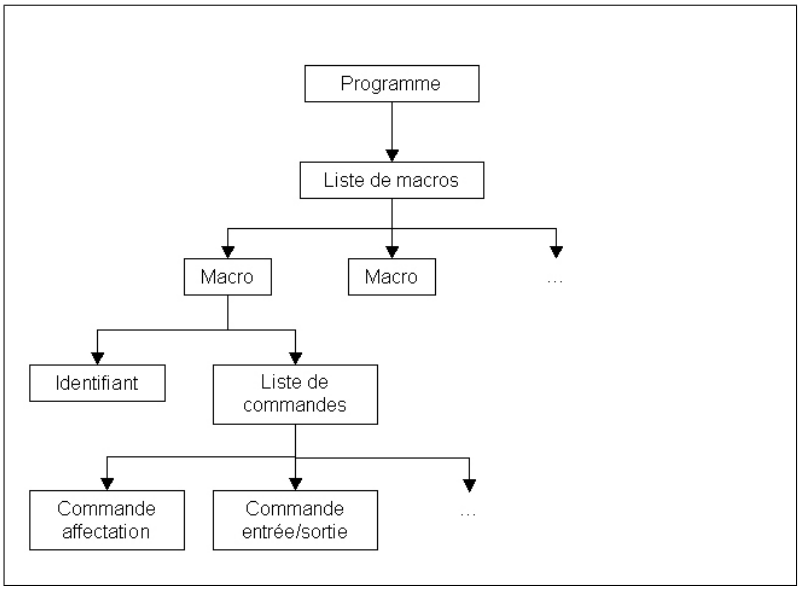

FIG. 4 – Hiérarchie d'objets résultant d'une analyse syntaxique

Par conséquent, une classe JAVA a été créée pour **chaque** règle de la grammaire de CSP, l'ensemble de ces classes étant contenu dans le package `compilateur.cspLangage`.

5.3 Génération de code

Comme ceci a été expliqué précédemment, à chaque règle syntaxique a été associée une classe JAVA. La génération de code a donc consisté à implémenter, pour chacune de ces classes, une méthode `compile` retournant le code JAVA associé à la règle CSP sous forme d'une chaîne de caractères.

Par exemple, la méthode `compile` de la classe `Affectation` pourrait être implémentée de la manière suivante :

```
public String compile () {
   return variableCible.compile() + "=" + expressionSource.compile ();
}
```

Par conséquent, la génération de code lié à un programme CSP consiste à appelé la méthode `compile` de l'objet `Programme`, résultat de l'analyse syntaxique.

6 L'interface

L'interface doit permettre à l'utilisateur une maitrise facile du compilateur et du déroulement de l'exécution. Ainsi, L'interface se compose de deux parties :

1. une partie visant à simplifier l'utilisation du compilateur :

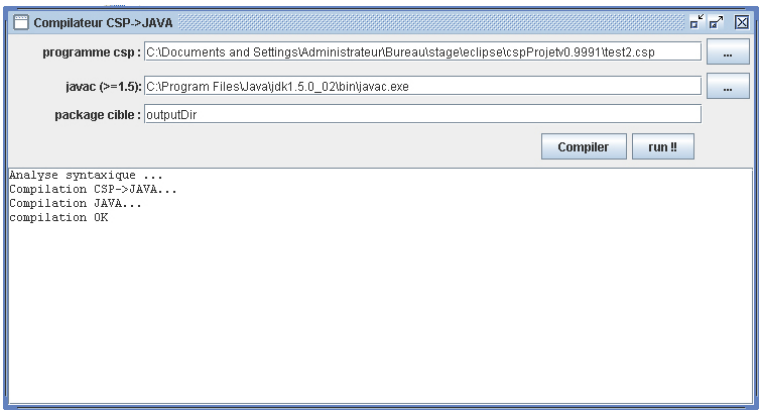

FIG. 5 – Fenêtre du compilateur

2. une partie *runtime*, permettant la manipulation et la visualisation de l'exécution du programme CSP en temps réel :

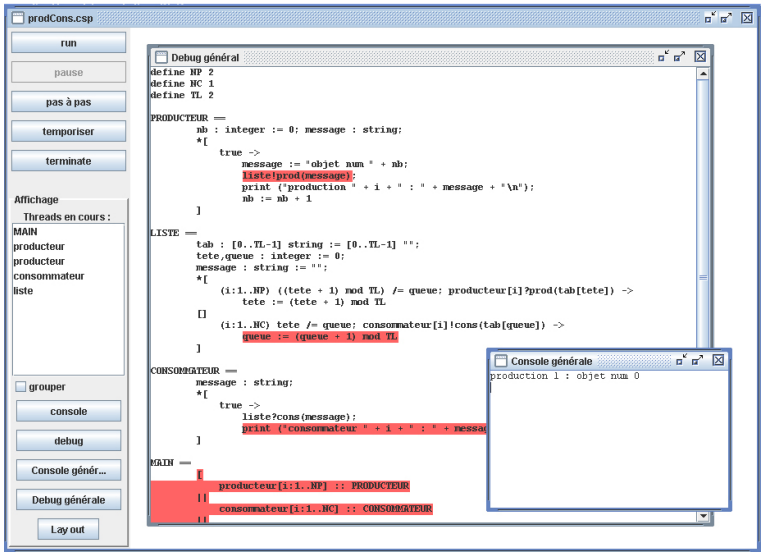

FIG. 6 – Fenêtre d'exécution

L'utilisation de l'une et l'autre de ces parties a été explicitée dans un manuel d'utilisateur fourni en annexe. Cependant, il est important d'apporter quelques précisions sur la deuxième partie, le suivi et le contrôle de l'exécution n'étant pas simple à implémenter.

6.1 Suivi du déroulement

Rappelons que la classe `CspThread` contient une méthode `print` (cf annexe ...), permettant à chaque processus d'afficher un message à l'écran. Afin de capter les appels à cette méthode tout au long de l'exécution du processus, un mécanisme de listeners a été implémenté.

Ainsi, à chaque fois qu'un processus souhaite afficher un message, les objets (ici la fenêtre d'interface) étant venu se référencer sur sa méthode `print` sont avertis et reçoivent le message à afficher.

Le suivi de l'exécution de chaque commande est quant à lui légèrement plus complexe, faisant intervenir le compilateur. En effet, c'est ce dernier qui a la charge d'ajouter **avant chaque commande** un appel à la méthode `debug`. Cette méthode, implémentée dans la classe `CspThread`, permet au processus d'indiquer la prochaine commande qu'il va exécuter. Ainsi, de même qu'avec la méthode `print`, un mécanisme de listeners permet à l'interface de détecter chaque appel à cette méthode, récupérant alors la position courante de l'exécution.

6.2 Contrôle du déroulement

Il est assuré au travers de la méthode `debug`. En effet, cette méthode est appelée avant chaque commande, et est chargée de dispatcher cet appel vers tous les listeners associés à l'objet. Le principe est donc d'ajouter un listener ayant pour seule utilité de dévier l'exécution du processus vers un *contrôleur d'exécution*, ici implémenté par la classe `Sync`.

Précisons alors que cet objet `Sync` peut fonctionner sous deux modes :
- un premier mode `run` laissant libre l'exécution de tous processus ;
- un second mode `pause` forçant tous les processus à venir ce bloquer sur l'objet. Ici, c'est la méthode `wait` qui est utilisée pour interrompre l'exécution.

Grâce à ce contrôleur d'exécution, quatre modes d'exécution sont possibles :

1. le mode **run** : il consiste à laisser libre l'exécution de tous les processus, configurant l'objet `Sync` en conséquence, et libérant (à l'aide de la méthode `notifyAll`) les éventuels processus ayant été interrompu auparavant ;
2. le mode **pause** : il consiste à interrompre tous les processus, configurant l'objet `Sync` en mode pause ;

3. le mode **pas à pas** : il a pour principe d'obliger chaque processus à n'exécuter qu'une commande à la fois. Pour ce faire, l'objet `Sync` est configuré en mode pause, les processus sont alors libérées par `notifyAll`, puis reviennent se bloquer sur le contrôleur d'exécution.
4. Enfin, le mode **temporiser** : il correspond à un mode pas à pas, à la différence que ce n'est plus l'utilisateur qui se charge de libérer les processus, mais un objet `Tempo` qui appelle la méthode `notifyAll` à intervalle régulier.

Pour finir, précisons que l'utilisateur a aussi la possibilité d'arrêter l'exécution du programme à l'aide d'un bouton `Terminate`. Ce fonctionnement est possible grâce à l'utilisation d'un booléen `mustStop` présent dans tous les processus dérivant de la classe `CspThread`, et qui est retourné à chaque appel de la méthode `debug` (permettant ainsi de vérifier régulièrement son état). Tout processus a donc pour contrainte de se terminer dès que cette méthode retourne `vrai`.

7 Bilan

7.1 Détails de la livraison

Voici donc concrètement ce qui a été fourni à la fin du stage :

- un compilateur de programmes CSP vers JAVA, avec son interface ;
- un manuel d'utilisateur explicitant le fonctionnement de cette interface ;
- le cahier des charges de l'application ;
- ce rapport et ses annexes.

7.2 Comparaison travail attendu / travail effectué

Globalement, le cahier des charges a été rempli : un compilateur de programmes CSP en JAVA a bien été réalisé, de même qu'un contrôleur pour gérer les communications. L'interface a cependant été plus poussée que prévu, le contrôle de l'exécution n'étant pas abordé dans le cahier des charges.

Notons cependant que le compilateur ne gère pas d'autres types que `integer`, `boolean` et `string`, le manque de temps n'ayant pas permis l'implémentation de types supplémentaires. Par ailleurs, le compilateur ne vérifie pas toutes les contraintes qu'il devrait pour un programme CSP. Par exemple, si dans une opération un booléen est additionné à un entier, le compilateur ne détectera pas l'erreur (par contre, l'erreur sera bien évidemment détectée par le compilateur JAVA).

7.3 Planning prévisionnel / planning réel

Sur le diagramme ci-dessous apparaissent planning prévisionnel (en rouge) et planning réel (en orange) :

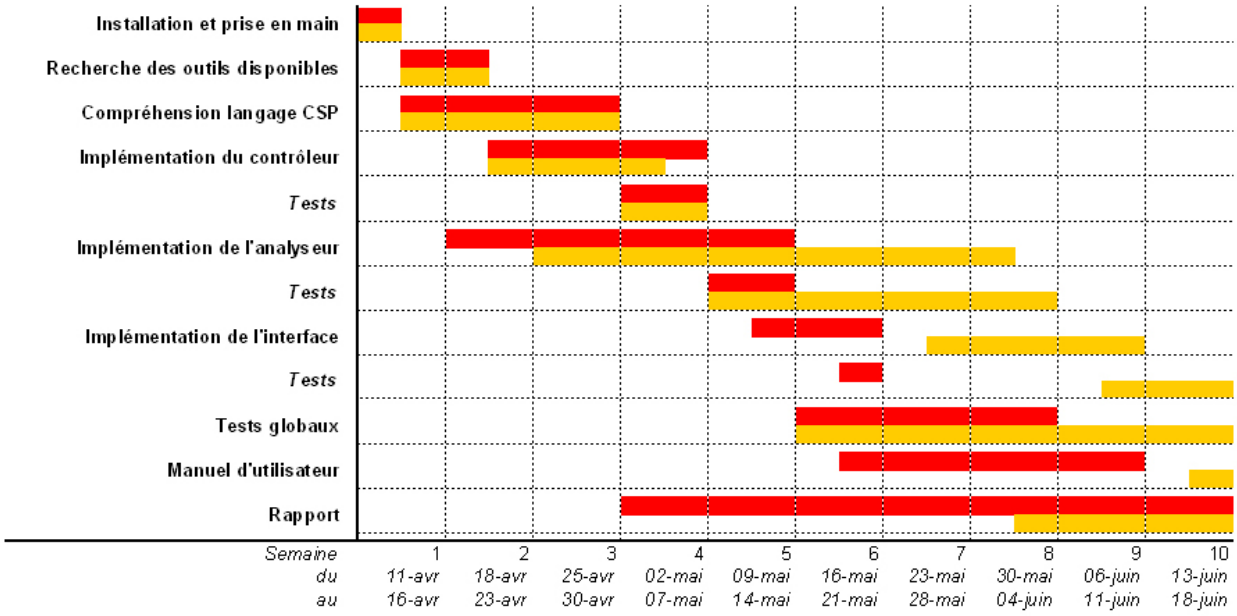

FIG. 7 – Comparaison planning prévisionnel / planning réel

On remarque donc de nombreuses différences, notamment concernant les phases d'analyse et de génération de code. En effet, la compilation reste un thème très peu abordé durant la formation de DUT Informatique, et c'est un principe qui s'est avéré bien plus complexe qu'il n'y paraissait.

Le contrôleur s'est quant à lui révélé plus simple, le traitement des commandes une à une permettant d'éviter de nombreux problèmes.

7.4 Évolutions possibles

Le compilateur fournit ne gérant pas d'autres types que `integer`, `boolean` et `string`, une première évolution serait d'étendre sa portée à des types comme `float` ou `double`.

Ensuite, une autre évolution serait d'améliorer la détection des erreurs du compilateur, notamment en ce qui concerne la vérification du typage dans les opérations et les affectations.

La partie contrôle de l'exécution de l'interface pourrait elle aussi être améliorée, en apportant notamment la possibilité de visionner la valeur de chaque variable.