# Application de CEYX à la construction de programmes sous forme de machines virtuelles

Matthieu Devin, Eric Madelaine, Annie Ressouche

INRIA

CENTRE
SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports de Recherche

N° 408

# APPLICATION DE CEYX À LA CONSTRUCTION DE PROGRAMMES SOUS FORME DE MACHINES VIRTUELLES

Matthieu DEVIN
Eric MADELAINE
Annie RESSOUCHE

Mai 1985

# Application de CEYX à la Construction de Programmes sous forme de Machines Virtuelles

**Matthieu Devin**
**Eric Madelaine**
**Annie Ressouche**

INRIA
Centre de Sophia-Antipolis
Route des Lucioles
06560 Valbonne

**Résumé**: Nous présentons des exemples d'utilisation du langage orienté objet CEYX à la conception et réalisation de programmes sous la forme de réseaux de machines virtuelles. Cette méthodologie apporte beaucoup de souplesse et de puissance pour une grande facilité de programmation.

**Abstract**: We present some examples of the object oriented language CEYX: we write programs as nets of virtual machines, in order to obtain very modular and dynamic systems. This method is flexible and powerfull, and makes programs easy to write.

**Mots Clés**: Langage orienté objet, CEYX, Lisp, Machine virtuelle, Architecture de programme, Programmation.

# Application de CEYX à la Construction de Programmes sous forme de Machines Virtuelles

## 1 INTRODUCTION

L'utilisation de Lisp pour écrire des systèmes expérimentaux complexes est de plus en plus fréquente. Elle permet d'écrire des programmes très modulaires, et d'obtenir rapidement des maquettes. Lors de la réalisation des systèmes CDS [Berry83a], Meije [Austry83a], Ecrins, nous avons utilisé CEYX [Hullot84a], qui nous apporte des possibilités de modularité différentes, plus proches de nos spécifications. Nous avons été amenés à concevoir ces systèmes sous forme de réseaux de machines virtuelles, communiquant par messages. Cette conception est assez proche de celle des acteurs de Plasma [Durieux81a]. Ces machines et ces messages s'implantent naturellement sous forme d'objets et de fonctions sémantiques CEYX. L'un des problèmes délicats rencontré est celui des machines d'entrée-sortie des systèmes (c'est un aspect souvent négligé hélas dans les logiciels de recherche, car il demande un investissement assez important, dans un domaine qui n'est pas l'objectif immédiat du système). Nous avons obtenu, dans des cadres variés, des machines d'entrée-sortie puissantes et faciles à utiliser. La même méthodologie est appliquée à l'ensemble de l'architecture des systèmes, permettant une conception et une réalisation aisée et rapide.

Le premier chapitre est une présentation rapide de CEYX; nous y présenterons les notions d'objet, de sémantique et d'héritage sémantique.

Nous présentons dans le deuxième chapitre un exemple d'architecture à base de machines virtuelles: celle du système CDS. On y verra comment on peut faire coexister deux analyseurs syntaxiques, à l'aide d'un brasseur de parsers, ainsi

qu'un premier exemple de machine d'entrée.

Le troisième chapitre présente l'architecture dynamique de la machine d'entrée du système ECRINS. elle se présente sous la forme d'un réseau de modules, reconfigurable dynamiquement et de façon décentralisée (aucun module, à un instant donné, ne connait la configuration de la machine). On y détaille un exemple de changement de configuration, et on discute du problème du maintien de la cohérence de l'architecture.

## 2 PRESENTATION DE CEYX

En CEYX les objets sont le résultat de la combinaison d'une structure de "record", et d'un ensemble de *comportements* (les *sémantiques*). La structure de "record" définit l'information contenue dans l'objet. Les sémantiques indiquent quelle est la réaction de l'objet à la réception de certains messages. Les objets sont organisés hiérarchiquement selon un arbre de types (classes dans la terminologie CEYX) selon lequel champs et comportements sont hérités.

Les noms de classes sont soit des atomes Lisp (Foo, Bar) soit des couples de noms de la forme {<sur-classe>}:classe ({Foo}:Bar est la sous-classe Bar de la classe Foo).

**(DEFTCLASS <nom-de-classe> <champ1>~<type1> ... <champN>~<typeN>)**

Définit une nouvelle classe de nom <nom-de-classe>. Les objets de cette classe possèdent les N champs <champI> qui contiennent des objets des classes <typeI>. Si <nom-de-classe> est une sous-classe les champs <champI> sont des champs supplémentaires à ceux hérités des sur-classes.

```
(deftclass Solide poids volume)     ; les solides ont un poids et un volume
(deftclass {Solide}:Cube arete)     ; la sous-classe des Cubes a un champ
                                    ; arête supplémentaire
```

**(OMAKEQ <classe> <champ1> <val1> ... <champN> <valN>)**

Rend en valeur un nouvel objet de la classe <classe> dont les champs <champI> ont les valeurs <valI>. On peut accéder aux champs de cet objet en lecture/écriture par les fonctions Lisp {<classe>}:<nom-du-champ>.

```
? (setq X (omakeq Cube poids 12))     ; On crée un Cube de poids 12
= #(#:Solide:Cube . #[12 () ()])
? ({Solide}:poids X)                  ; Son poids est bien 12
```

```
= 12
? ({Cube}:arete X 3)                    ; Le champ arete est
?                                       ; rempli par 3
= #[12 () 3]
? ({Cube}:arete X)                      ; Ce que l'on vérifie
= 3
```

Les sémantiques (comportements) associés aux classes sont des fonctions Lisp usuelles dont le nom a la forme {<classe>}:<message>. Lors de la transmission du message <m> à un objet de la classe <c> on évalue la fonction {<c>}:<m> avec comme premier argument l'objet lui-même et comme arguments supplémentaires les éventuels paramètres passés avec le message.

**(DE {<classe>}:<message> (objet par1 ... parP) . <corps>)**

Définit la fonction Lisp qui est évaluée lors de la transmission du message <message> à un objet de la classe <classe>. Le corps de la fonction ainsi définie est un corps de fonction Lisp classique: toute la puissance Lisp est donc disponible en CEYX.

**(SEND <message> <objet> <par1> ... <parN>)**

Transmet un message avec les paramètres <parI>.

```
? ; Définissons la sémantique "densité" des Solides.
? (de {Solide}:densite (objet)
?     (/ ({Solide}:poids objet) ({Solide}:volume objet)))
= #:Solide:densite
? ; Appliquons la à un Solide
? (send 'densite (cmakeq Solide poids 12 volume 3))
= 4
```

L'héritage sémantique est compris ainsi: les objets d'une sous-classe savent répondre au moins aux mêmes messages que les objets de leur sur-classe. Dans notre exemple les Cubes savent répondre au message *densite* en appliquant la sémantique *densite* de leur sur-classe *Solide*.

```
? (send 'densite (cmakeq Cube poids 20 volume 2))
= 10
```

## 3 L'ARCHITECTURE DE Cds

Cds est un langage de programmation fonctionnel développé et implémenté par l'équipe EQDOM dans le cadre d'un projet du GRECO de programmation [Berry83a]. Nous présentons ici les principes de programmation qui ont guidés son implantation dans le langage orienté objet Ceyx [Hullot84a].

Les problèmes posés par cette implantation tiennent plus de l'architecture de programmes que de l'algorithmique. En effet le modèle mathématique du langage fournit une sémantique sous forme de règles de réécriture dont l'implantation est immédiate. Les problèmes d'entrées/sorties, d'analyse syntaxique du langage (qui possède ici deux syntaxes alternatives), et d'interface système (timer, exécution de commandes systèmes, etc.) sont par contre assez cruciaux pour exiger la réalisation d'outils permettant une implantation propre et modulaire, réutilisable pour d'autres langages.

Dans cette optique nous avons construit l'interprète à partir de *machines virtuelles* : des objets autonomes réalisant les actions de base telles que l'analyse lexicale, l'analyse syntaxique, le calcul des expressions, etc.. Nous précisons d'abord la notion de machine virtuelle, montrant comment celles-ci s'implémentent de manière naturelle dans le langage orienté-objet Ceyx. Nous explicitons ensuite les différentes machines qui ont servi à la réalisation de l'interprète Cds.

### 3.1 Les machines virtuelles et leur implantation

#### 3.1.1 Description

Une **machine virtuelle** est un objet autonome qui sait réaliser diverses actions bien définies. Une machine peut être connectée à d'autres machines virtuelles avec lesquelles elle communique par messages.
Les machines virtuelles sont naturellement **typées**: les machines d'un type donné savent réaliser certaines actions particulières.
Les types de machines forment un arbre de types, du plus général au plus particulier: les machines d'un sous-type d'un type donné savent réaliser *au moins* toutes les actions du (sur-)type.
Un certain nombre de **champs** permettent de décrire l'état interne d'une machine et ses connections avec le monde extérieur.

Une machine virtuelle peut, en un certain sens, être perçue comme un **acteur**

Plasma. Le mécanisme de transmission de messages que nous utilisons pour faire communiquer les machines est cependant plus frustre que celui de Plasma: il n'y a pas de notion de continuation, et exige une exécution **séquentielle** des actions. Ce dernier point est particulièrement important pour la réalisation de l'architecture dynamique décrite plus loin.

### 3.1.2 Implémentation

Ces machines s'implémentent naturellement dans un langage objet "classique", avec héritage de "comportements" (méthodes de smalltalk, sémantiques de CEYX) selon un arbre de "classes".

Les machines sont implémentées comme des objets CEYX. Les actions que peuvent réaliser les machines sont des sémantiques. La hiérarchie des classes CEYX correspond naturellement à la hiérarchie des types de machines: par héritage sémantique toutes les machines d'une sous-classe (d'un sous-type) répondent aux même messages (réalisent les mêmes actions) que les machines d'un type supérieur, c'est à dire moins spécialisé, dans la hiérarchie.

## 3.2 le Kit de base CDS

L'interprète CDS est bâti autour de plusieurs machines différentes:

- Un **calculateur** qui évalue les termes du langage.

- Une **machine d'entrée/sortie** qui s'occupe des interfaces "caractère" avec l'utilisateur et le système.

- **Des analyseurs syntaxiques** (ou parsers) qui construisent des expressions CDS à partir des caractères fournis par la machine d'entrée/sortie.

- Un **toplevel** qui exécute les commandes "systèmes", et gère le calculateur et la machine d'entrée sortie.

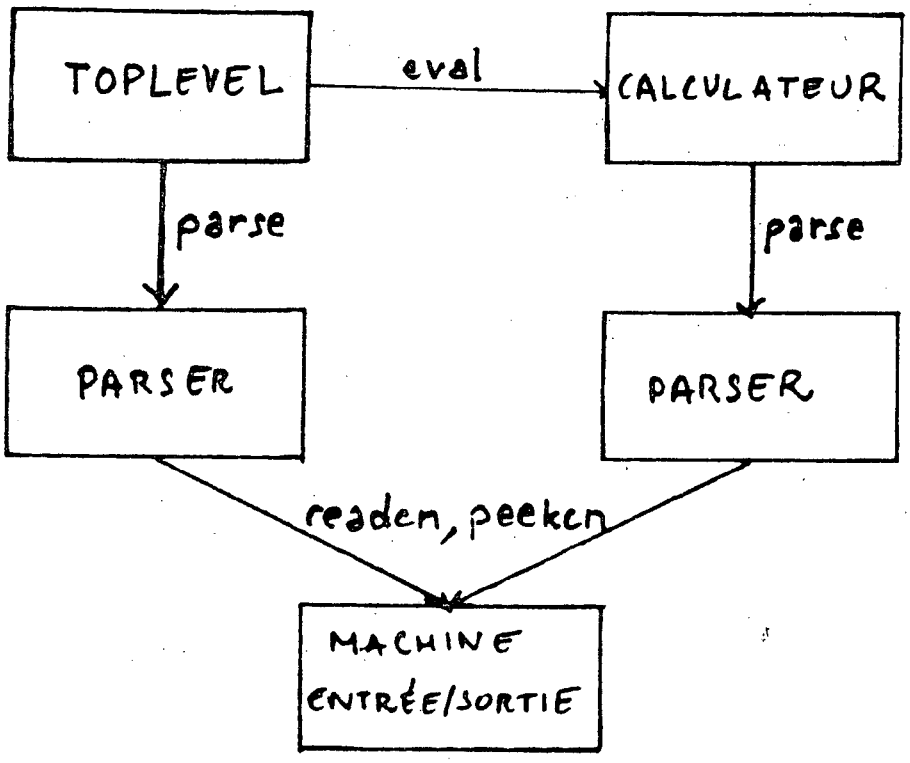
Le fonctionnement typique du système est le suivant:

1) le toplevel envoie le message *parse* à l'analyseur syntaxique.
2) l'analyseur envoie alors des messages de lecture de caractères (*readcn*, *peekcn*) à la machine d'entrée/sortie pour reconnaître une expression CDS.
3) l'analyseur répond au message *parse* en rendant cette expression CDS.
4) le toplevel envoie le message *eval* avec pour paramètre cette expression au calculateur, qui effectue alors l'évaluation du terme.

5) le calculateur évalue l'expression. En CDS l'évaluation est dirigée par l'utilisateur; le calculateur a donc besoin d'un parser pour recevoir les "directives" lui permettant d'évaluer le terme. On rentre donc dans un autre dialogue *parse, readcn-peekcn*.

6) le calculateur envoie le résultat de l'évaluation à la machine d'entrée/sortie pour impression.

On peut schématiser le réseau de machines, avec les messages transmis comme suit:




Il est intéressant de voir plus en détail l'implantation des parsers et de la machine d'entrée sortie.

### 3.2.1 Les parsers

Le type de machine "parser" est défini comme une machine sachant répondre au message *parse*. Les deux machines qui ont besoin de parsers (le toplevel et le calculateur) disposent d'un champ "parser" contenant un objet de type "parser" auquel elles envoient le message *parse*.

L'implantation de ces machines requiert donc la déclaration d'une classe des parsers et de leur sémantique *parse*, puis la déclaration des types de machines toplevel et calculateur.

```
(DEFTCLASS Parser langage~Symbol)   ; Les parsers ont un nom qui indiquent
                                    ; le nom du langage
(DE {Parser}:parse (p))

(DEFTCLASS Toplevel parser~Parser ...)
(DEFTCLASS Calculateur parser~Parser ...)
```

Il faut maintenant déclarer les parsers eux-mêmes. La bibliothèque CEYX fournit un générateur d'analyseur lexicaux: CxYacc*. Les parsers générés par CxYacc sont de la classe *{Parser}:CxYacc*. Ils disposent d'un analyseur lexical (de la classe LexKit), qui leur fournit des tokens construits à partir de caractères puisés dans une machine d'entrée.
La classe CxYacc peut être définie par:

```
(DEFTCLASS {Parser}:CxYacc lexical~LexKit ...)
```

Nous ne décrivons pas la sémantique *parse* qui lui est propre.

Le langage CDS dispose de deux syntaxes alternatives, on a donc besoin de deux parsers distincts, en amont du toplevel. La connection de ces deux parsers au toplevel est réalisée par un *brasseur de parsers*. Ce brasseur est d'un nouveau type de parsers, connaissant un ensemble de parsers distincts, dont un seulement est actif à un moment donné. Il transmet le message *parse* au parser actif, et sait changer de parser actif au message *brasse*.

Ce nouveau type de machine est défini come suit:

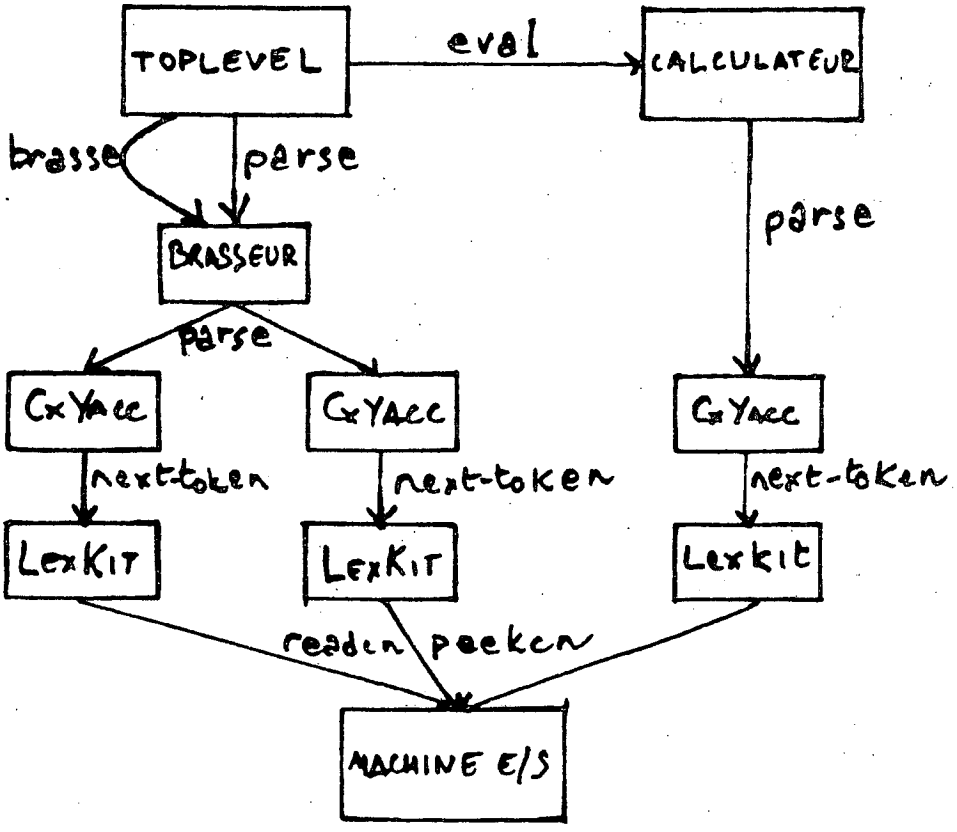
```
(DEFTCLASS {Parser}:Brasseur parser-actif~Parser parsers~(List Parser))

(DE {Brasseur}:parser (p)
    (SEND 'parse ({Brasseur}:parser-actif p)))

(DE {Brasseur}:brasse (p langage) ...)
```

*Remake de Yacc qui génère des parsers en CEYX.

Avec ces nouveaux parsers l'architecture du système devient:




Il est intéressant de constater que le toplevel ne sait pas a-priori de quel type exact est l'objet de son champ *parse*. Il lui suffit de savoir que cet objet est une machine de type *Parser* pour lui envoyer les messages *parse*. Dans notre cas le toplevel est effectivement connecté à un brasseur et non à un objet de type CxY-acc.

### 3.2.2 La machine d'entrée/sortie

La machine d'entrée/sortie doit réaliser les tâches élémentaires d'impression du caractère d'invite ("prompt" dans la suite), d'écho éventuel des caractères lus, de sélection de divers flux de caractères (clavier, fichiers), et d'arrêt sur certains caractères (dans le buts de programmes de démonstrations automatiques).

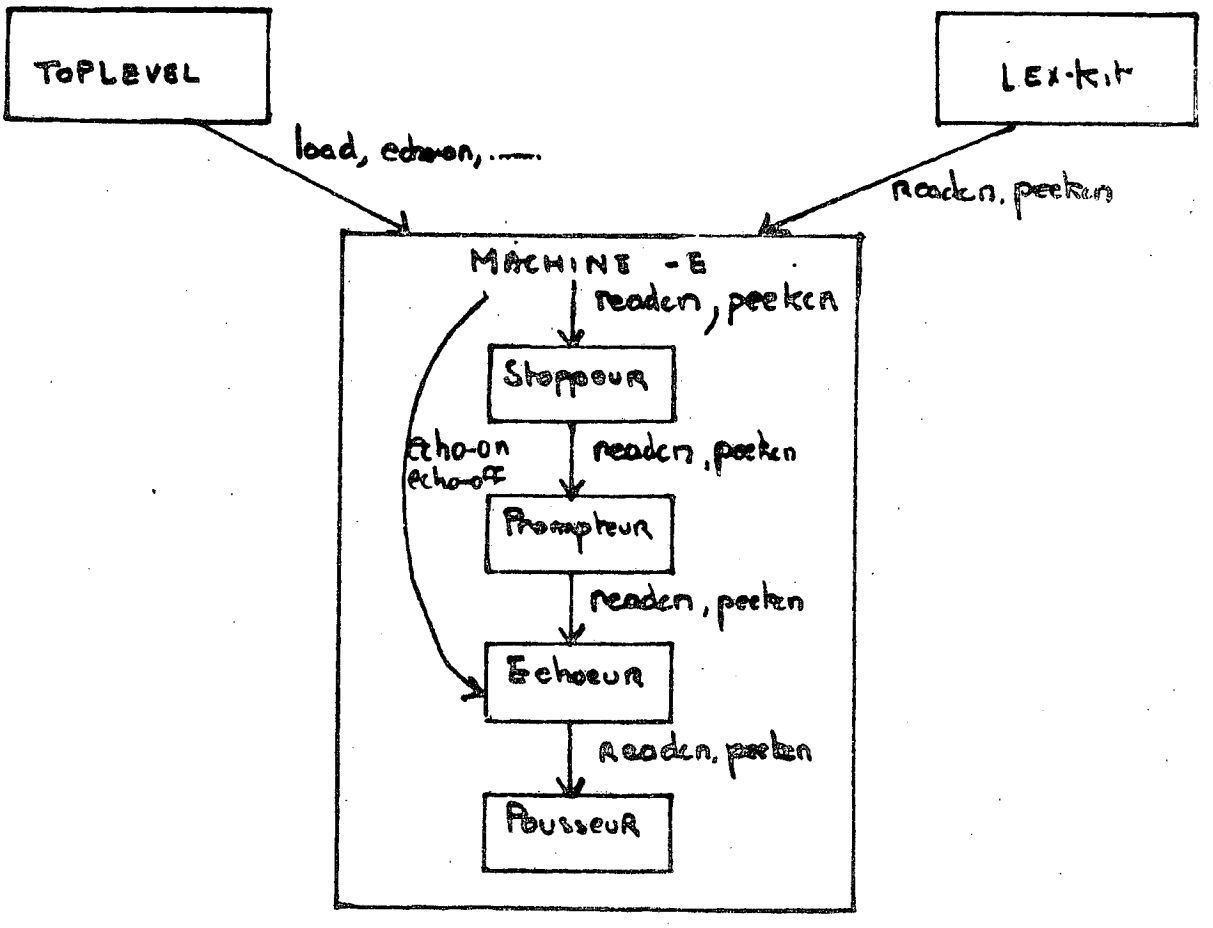
Cette machine est constituée de plusieurs machines élémentaires réalisant chaque tâche indépendemment. Ces machines sont toutes du type *Machine-E* des machines d'entrées qui garanit une action à la réception des messages *readcn* et *peekcn*.

```
(DEFTCLASS Machine-E)
(DE {Machine-E}:readcn (m))
(DE {Machine-E}:peekcn (m))
```

Les machines spécifiques réalisant l'écho, le prompt, etc. disposent de champs supplémentaires décrivant leur état (doit on prompter, doit on faire l'écho, ...). Ces machines sont connectées en pipe-line: le flux de caractères obtenus d'un "lecteur" (qui réalise l'interface physique avec LELISP) traverse chaque machine en série. Les sémantiques *readcn*, *peekcn* sont redéfinies pour effectuer le traitement nécessaire sur le caractère obtenu de la machine "en amont" dans le pipe-line.

La Machine d'entrée du système CDS est encore un nouveau type de machine, formé d'une collection de ces machines élémentaires. Les messages de lecture sont transmis à la première machine du pipe-line, et les messages de gestion des machines élémentaires (faire/ne pas faire l'écho, ...) sont transmis directement à la machine concernée.

La machine d'entrée, peut être ainsi schématisée:



### 3.3 Conclusion

Les techniques classiques de conception de programmes amènent à découper les systèmes en "boîtes noires" réalisant des tâches élémentaires. Le système est alors considéré comme un réseau dont les noeuds sont ces boîtes, et les arêtes des moyens de communication entre boîtes. La technique de programmation sous forme de machines virtuelles est similaire: les boîtes noires sont des objets d'un langage qui communiquent par messages. Un langage orienté objet permet d'implémenter ces réseaux de manière naturelle.

La hiérarchie de types proposée par CEYX, permet de réaliser des machines indépendantes du contexte de leur utilisation: on peut effectivement mettre en place d'un machine d'un certain type toute machine d'un type inférieur, c'est à dire plus spécialisé, sans qu'il soit nécessaire d'en informer les autres machines du réseau.

L'architecture que nous avons décrit est cependant complètement **statique**: Le réseau est construit au lancement du système et reste figé durant tout le calcul. C'est lors de la construction initiale que l'on peut effectivement connecter les machines différemment, ajouter des machines (on pourrait facilement ajouter une machine pour traiter les messages d'erreur), ou en retrancher (on pourrait n'utiliser qu'un seul parser pour le toplevel).

Nous considérons que le jeu de machines virtuelles que nous avons construit forme un LEGO® de machines qui peuvent être assemblées différemment selon les fonctionnalités requises. Notons que ces machines ont étées réutilisées telles quelles, dans un assemblage peu différent, pour réaliser l'interprète du langage ESTEREL [Berry84a] développé par l'équipe Parallélisme et Synchronisation à Sophia-Antipolis.

## 4 ARCHITECTURE DYNAMIQUE

La machine d'entrée de CDS a une architecture statique. Elle comprend des modules habituellement non utilisés, mais qui peuvent être activés par des messages *(prompt-on, prompt-off, echo-on,... )*. Ceci reste assez proche des techniques de programmation classique, pour lesquelles il faut prévoir toutes les configurations possibles, et un jeu de fonctions ou d'indicateurs booléens pour contrôler le comportement du programme.

Nous allons maintenant proposer une architecture beaucoup plus souple, où

les modules pourront être non plus activés, mais créés dynamiquement, soit par un utilisateur interactif, soit par d'autres modules. Le contrôle de l'architecture est dynamique dans le sens où l'on peut (par des messages) insérer des modules, en changer les liaisons, détruire des modules. Nous avons utilisé cette méthode pour construire la machine d'entrée du système ECRINS (système de manipulation et de preuve pour les calculs de processus).

### 4.1 Buts poursuivis

Ils sont doubles: efficacité d'une part et souplesse d'autre part. Dans l'architecture statique de la machine de CDS, la présence de modules inactifs dans certaines configurations entraine une certaine inefficacité. En maintenant à tout instant une architecture minimale (c'est à dire ne comportant pas de modules inutiles au fonctionnement courant), on peut espérer obtenir une meilleure efficacité. Nous verrons cependant que les techniques utilisées pour réaliser cette architecture dynamique ne permettent pas, dans la plupart des cas, de gagner cette efficacité. Nous proposerons, dans la conclusion, une méthode totalement différente permettant d'avoir des machines beaucoup plus efficaces, mais seulement dans le cas statique.

L'autre motivation est la souplesse. La machine d'entrée de CDS convient parfaitement à un système "fermé", où le concepteur a figé une fois pour toutes les fonctionnalités voulues. Dans le système ECRINS, par contre, nous voulons pouvoir ajouter à n'importe quel moment des éléments dans la machine d'entrée, et même donner à l'utilisateur du système un contrôle sur son architecture (ou au moins à certains utilisateurs privilégiés). Nous avons donc choisi une architecture dynamique, sans contrôle centralisé: un certain nombre de modules, extérieurs à la machine d'entrée, sont susceptibles d'en modifier l'architecture, mais aucun, à un instant donné, ne connait complètement la configuration. Il leur suffit de connaître les éléments qu'ils ont eux-même introduits. Nous discuterons plus bas les problèmes de cohérence que cela pose.

### 4.2 Filtres et Messages

La machine d'entrée est conçue essentiellement comme un pipe (au sens Un*x): les modules sont des filtres qui traitent certains des messages qu'ils reçoivent et en transmettent d'autres. A tout instant, la partie active de la machine est une chaine linéaire de filtres, liés par leur champ *in-stream*, capables de convoyer des messages de type *readcn*, *peekcn*,... jusqu'au lecteur situé en bout de chaine, de traiter le cas échéant l'information portée par ces messages, et de retourner le résultat vers le module appelant.

Le chainage des modules est réalisé à travers un champ particulier des objets de type Filtre: le champ *in-stream*. Pour chaque module ce champ contient le filtre suivant dans la chaine; l'envoi d'un message "mess" par un filtre "filtre" au

maillon suivant de la chaine s'écrit donc:

```
(send 'mess ([Filtre]:in-stream filtre) args...)
```

Nous distinguerons deux sortes de filtres, selon le traitement qu'ils effectuent sur le flux de caractères:

-- Les filtres "actifs" traitent leur flux de caractères, soit pour le modifier (stoppeur), soit pour en faire un écho, soit pour exécuter des actions particulières sur certains caractères (stoppeur, prompteur).

-- Les filtres "passifs" transmettent les caractères sans même les regarder.

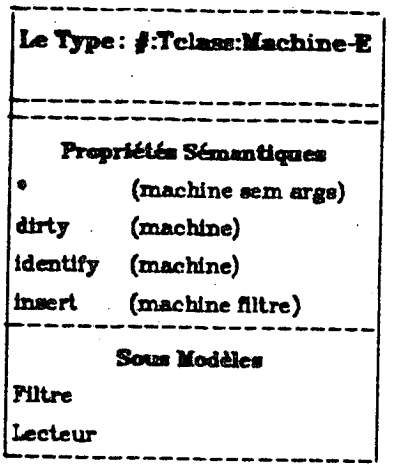
Les messages qui vont circuler dans la chaine de filtres sont de trois types:

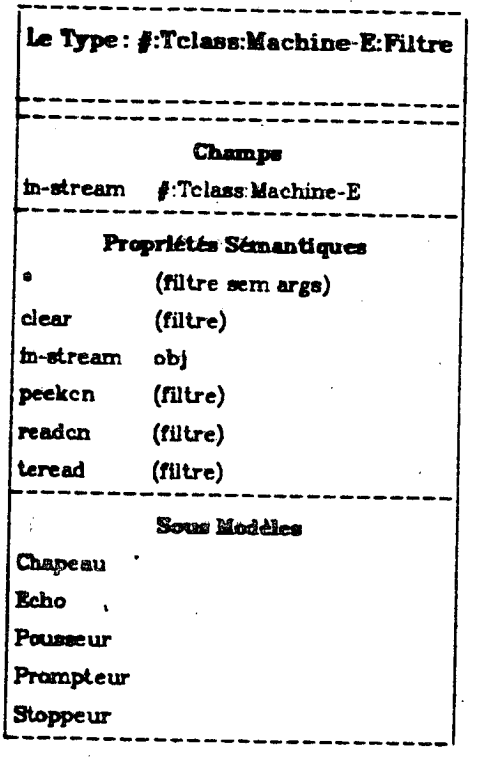
-- Les messages de configuration de l'architecture. Ils sont capables d'insérer de nouveaux modules dans la chaine ou d'en détruire. Cela pose des problèmes d'adressage dont nous verrons plusieurs solutions.

-- Les messages spécifiques à certains modules. Ils doivent être transmis jusqu'à leur cible indépendamment du type des filtres placés avant. Nous utiliserons pour cela un mode spécial de propagation des messages, appelé "sémantique *": Imaginons un message * *foo* arrivant sur la machine d'entrée; son premier fils, l'interface, n'a pas de sémantique *foo* (la fonction #:Machine-E:Filtre:Interface:foo n'est pas définie, ni #:Machine-E:Filtre:foo, ...). Plutôt que de provoquer une erreur *semantique inconnue*, le message est transmis tel quel sur le champ *in-stream* de l'interface, et ainsi de suite jusqu'à ce qu'il arrive à un filtre capable d'exécuter la sémantique *foo* (ou provoque une erreur en bout de chaine).

-- Les messages classiques des machines d'entrées *(readcn, peekcn, teread)* qui doivent être transmis jusqu'au lecteur courant. Ils constituent effectivement le "flux de caractères" qui traverse la machine d'entrée. Ils peuvent être ou ne pas être traités "au passage" par les filtres. On utilise ici le mécanisme d'héritage sémantique de CEYX. Par défaut on utilise les sémantiques *readcn, peekcn*, ... définies au niveau du type Filtre. Elles sont seulement redéfinies au niveau des filtres actifs, lorsqu'il y en a vraiment besoin. Par exemple:

```
(defmethod [Filtre]:readcn (filtre) (in-stream) (send 'readcn in-stream))
(defmethod [Echo]:readcn (echo) (in-stream machine-s)
   (let ((car (send 'readcn in-stream)))
        (send 'princn machine-s car)
        car)))
```

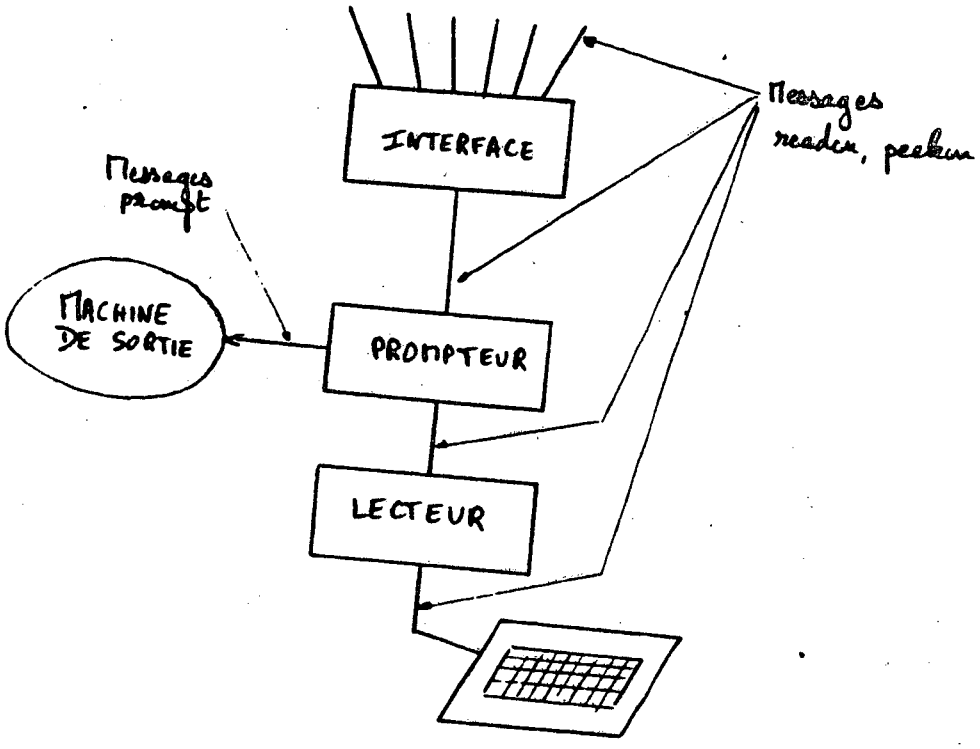



### 4.3 Exemples

Donnons maintenant un exemple de reconfiguration dynamique de la machine d'entrée. Pendant une session interactive sous le système ECRINS, la machine d'entrée est habituellement dans une configuration simple, avec un seul filtre actif: le prompteur qui envoie au terminal (plus précisément à une machine de sortie) des prompts primaires ou secondaires à chaque fois que l'analyseur du langage de commande a lu un caractère CR (fin de ligne) et a besoin d'un

caractère supplémentaire.




Le module INTERFACE est un filtre passif particulier mémorisant l'ensemble des modules externes qui connaissent la machine d'entrée. Il permet dans certains cas de remonter des informations vers les modules utilisant la machine d'entrée. C'est l'interface "amont" de la machine.

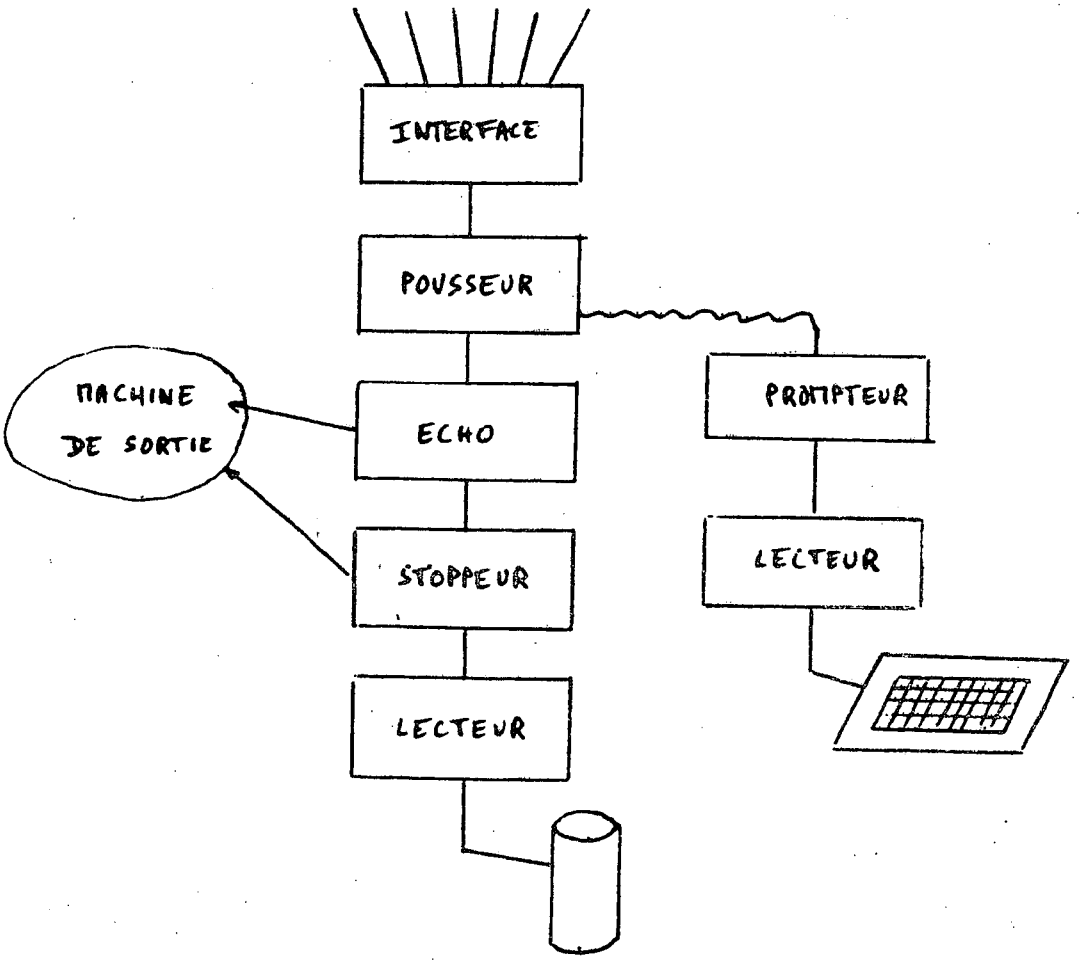
Le LECTEUR connait un canal de lecture LeLisp, en l'occurence celui associé au terminal.

Le PROMPTEUR est un filtre actif: il transmet au lecteur les demandes de caractères qui lui parviennent, et reconnait dans la réponse du lecteur les cas où il doit agir (C'est à dire envoyer à la machine de sortie le message *prompt*. C'est la machine de sortie qui selon la couleur associé à ce prompteur particulier, et selon les caractéristiques des organes de sortie courants - écran, fichier, bit-map - décidera du caractère de prompt à envoyer).

Observons maintenant notre utilisateur interactif. Il veut examiner le déroulement d'une démonstration figurant sur un fichier. Le fichier contient une séquence d'ordres, et (toutes les 10 à 15 lignes) des caractères d'arrêt. Le

système reconnaitra un caractère d'arrêt, interrompera le défilement de la démonstration, laissant à l'utilisateur le temps de la lire, jusqu'à ce que celui-ci ne frappe à son clavier le caractère commandant la reprise du défilement. Il lance son processus par une commande *load-demo* dont nous allons détailler le fonctionnement.

Il s'agit d'un changement de configuration de la machine d'entrée: la chaine de filtres courante va être mémorisée par un POUSSEUR (filtre passif), et remplacée par une chaine aboutissant au fichier. Celle-ci n'a pas besoin de prompteur, par contre il lui faut un module d'ECHO, pour envoyer au terminal une copie des expressions lues sur le fichier, et un STOPPEUR pour gérer les caractères d'interruption de la démo.




Le STOPPEUR reconnaissant un caractère d'arrêt interrompera la lecture, c'est à dire enverra à la machine de sortie un message d'impression (par exemple "Taper CR pour reprendre la démo") et attendra pour transmettre les prochaines sémantiques de lecture. Conceptuellement, le STOPPEUR est un filtre passif tant qu'il n'a pas reconnu de caractère d'arrêt. Ensuite il joue un rôle qui

n'est plus celui d'un filtre, mais d'un module maître, pour une machine d'entrée réduite au lecteur de clavier, et qui réalise localement l'interprétation du flux de caractères (qu'elle ne transmet d'ailleurs pas, y compris le caractère de reprise).

Comment s'est passé le changement de configuration?

L'utilisateur (ou le module interprétant ses commandes) a envoyé à la machine d'entrée sortie le message d'insertion suivant:

```
(send 'insert machine-e
              (cmakeq Pousseur))
```

Ce qui a créé le module POUSSEUR. Puis lui a connecté la machine d'entrée existante, c'est à dire la chaine de filtres sauf l'interface, qui est la seule partie immuable de la machine. Ou pour parler CEYX, a mis dans le champ *in-stream* du pousseur le prompteur qui était dans le champ *in-stream* de l'interface. Puis, de même, a connecté le pousseur sous l'interface.

Le message suivant est un message spécifique destiné au pousseur: c'est l'ordre d'empiler la nouvelle chaine a la place du fils courant du pousseur. Cela s'écrit:

```
(setq lecteur (cmakeq Lecteur chan (open! "file")))
(setq stoppeur (cmakeq Stoppeur char-stop >
                                char-restart CR
                                in-stream lecteur))
(setq echo (cmakeq Echo in-stream stoppeur))

(send '* machine-e 'pousser echo)
```

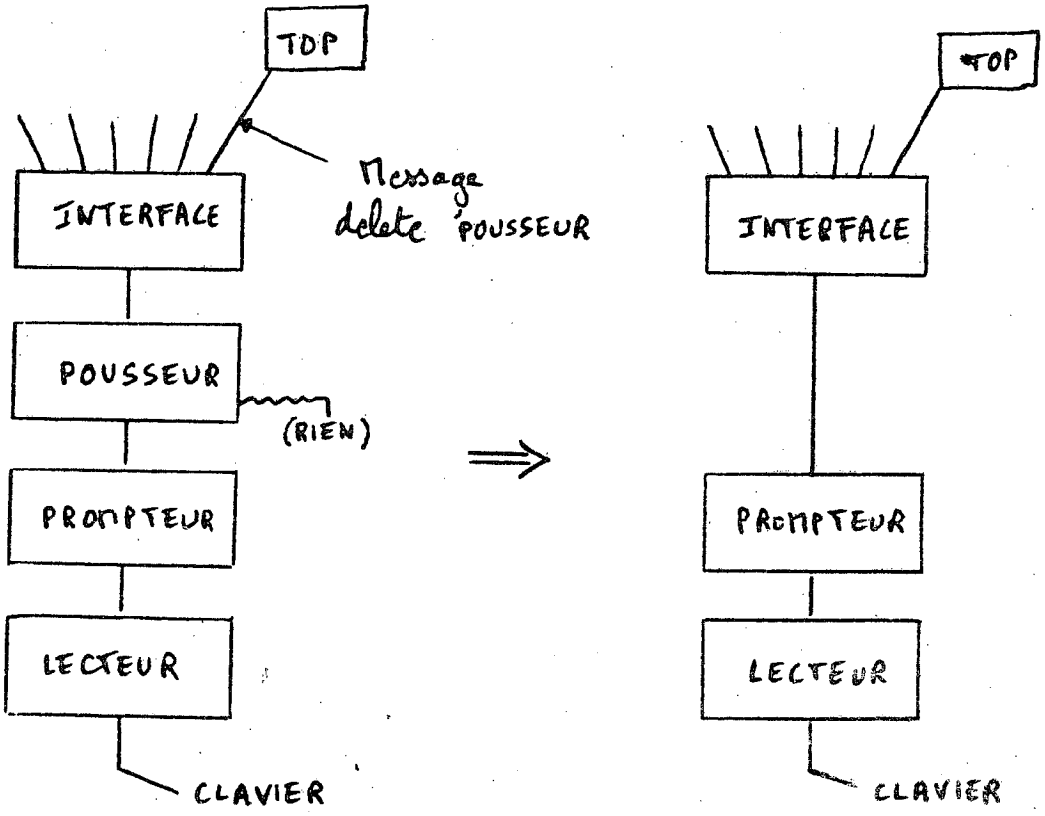
Ce message arrive au premier filtre de la chaine, l'interface. Celle-ci, n'ayant pas de sémantique *pousser*, le transmet à son fils, le pousseur. Celui-ci possède effectivement une sémantique *pousser*, et l'exécute.

Voyons maintenant comment on supprime un élément d'une chaine de filtres:

La lecture du fichier de démos terminée, le module qui interprète les commandes de l'utilisateur entreprend deux actions: il envoie au pousseur un ordre de dépilage (c'est à dire de restauration sous le pousseur de la chaine PROMPTEUR -- LECTEUR -- CLAVIER de la configuration de départ). Puis il envoie à la

machine d'entrée un ordre de suppression du filtre POUSSEUR, devenu inutile.

Nous avons choisi une implémentation où de tels messages de suppression s'adressent "au premier filtre du bon type". Le message de suppression porte comme adresse le type (au sens CEYX) de l'objet à éliminer. Ce message est propagé par la chaine de filtres (grâce à l'héritage de la sémantique {Filtre}:delete) jusqu'au père du POUSSEUR (ici l'interface). Celui-ci reconnaissant que son fils est du type désigné par le message le supprime, c'est à dire le remplace dans son champ *in-stream* par son petit-fils.




Notons que d'autres modes d'adressage sont envisageables, mais que le mode "au premier filtre du type" est le plus naturel dans notre cas.

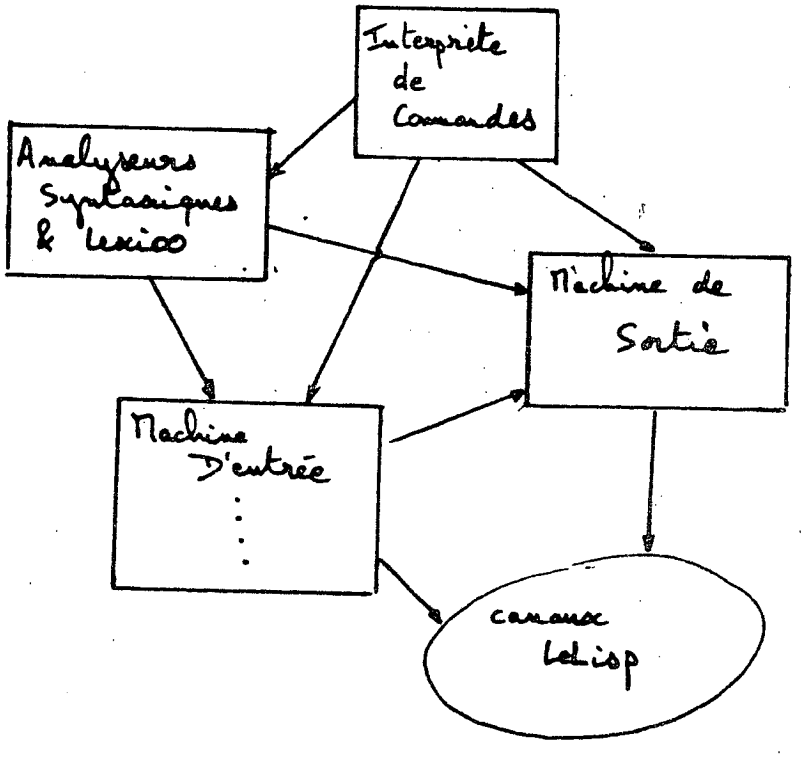
-- Il peut être intéressant d'envoyer des messages spécifiques "à tous les filtres du bon type"; cette fonctionnalité (réalisable sous la forme d'une sémantique {Filtre}:delete-all) peut se dériver de la précédante. De plus, son utilisation est peu fréquente dans le cas du système ECRINS, où l'on a rarement plusieurs filtres de même type.

- D'autres applications nécessiteront un adressage plus précis, utilisant des modules nommés. Cette méthode est peu compatible avec la philosophie décentralisée utilisée dans ECRIN, où aucun module ne joue le rôle d'un gestionnaire de l'architecture.

### 4.4 Maintien de la cohérence

Le problème de maintenir la cohérence d'un réseau de modules se reconfigurant dynamiquement, sans connaissance centralisée de l'état du réseau est en général difficile. Les conditions qui nous sont imposées par le langage d'implantation (asynchronisme et séquencialité: les messages sont réalisés par des appels fonctionnels) nous permettent de restreindre le problème.

Nous nous placerons dans le cadre d'une structure modulaire hiérarchique. Au niveau supérieur, on dispose d'une structure à peu près statique, chaque module étant une ressource utilisable par tous les autres (la machine de sortie en est un cas typique: elle est un module unique à travers de laquelle doivent passer tous les messages d'impression, quelque soit leur provenance). Certains des modules de ce niveau (et nous reprenons l'exemple de la machine d'entrée) sont eux-même compos'es de modules du niveau inférieur, liés par une architecture dynamique. Comment les modules extérieurs à cette machine d'entrée doivent-ils communiquer avec elle pour préserver sa cohérence?




Précisons d'abord ce que nous entendons par cohérence. Les "incidents" pouvant intervenir dans une structure dynamique sont de plusieurs espèces:

coupures de la chaine de filtres, persistance de modules déconnectés de la chaine (mais non récupérables par le ramasse-miettes LeLisp), persistance de modules passifs inutiles dans la chaine active, non-linéarité de l'architecture active,... La configuration de notre machine d'entrée sera dite **cohérente** si:
Elle présente une structure passive arborescente (et non pas de graphe), et une structure active (chaine liée par les champs *in-stream*) linèaire, aboutissant à un LECTEUR. De plus, aucun élément de cette chaine autre que l'INTERFACE ne doit être référencé depuis l'extérieur de la machine d'entrée.

Elle sera dite minimale si il ne figure aucun filtre passif inutile dans la chaine active (ceci est une notion qui demanderait à être plus formalisée, mais nous ne le ferons pas ici).

Les hypothèses qui nous permettront d'assurer un fonctionnement cohérent et minimal à notre machine d'entrée sont les suivantes:

- Tous les messages à destination de la machine d'entrée passent par l'interface. En particulier les messages de reconfiguration (*insert et delete*); ils ne peuvent donc agir que sur la chaine active.

- Les modules externes ne connaissent pas d'autres objets que l'interface. Ils ne doivent pas conserver de références directes sur les autres filtres, et n'ont aucun moyen d'en obtenir (pas de fonction qui "rend" une référence sur un filtre).

- Chaque module externe est chargé d'assurer la minimalité vis à vis des filtres passifs qu'il a lui-même introduit. Par exemple, si c'est le *top-level* qui a inséré un pousseur, il est le seul à pouvoir l'oter de la chaine; mais la minimalité sera meilleure si c'est l'interprete de commande qui introduit le pousseur lors de l'exécution d'une commande *load*, et qui l'enlève immédiatement après. Rappelons-nous que l'implémentation fonctionnelle des messages facilite un tel fonctionnement: la fonction *load* de l'interprète de commande peut être écrite:

```
(de {Interp}:load (filename)
    (send 'insert machine-e (cmakeq Pousseur))
    (send '* machine-e 'push (....))
    (send 'parse analyseur)
    (send '* machine-e 'pop)
    (send 'delete machine-e 'Pousseur))
```

Cette condition joue aussi un rôle vis à vis de la cohérence: aucun module externe ne doit pouvoir détruire des éléments qu'il n'avait pas lui-même introduit. En particulier le message *delete-all <type>* est interdit.

Ces conditions permettent de conserver la cohérence et un certain degré de minimalité en fonctionnement normal. Cependant certaines des hypothèses sont relatives au comportement de modules externes, donc éventuellement de l'utilisateur du système. Nous mettrons donc à sa disposition une commande *reset*, capable de remettre le système dans un état correct, comportant simplement un prompteur et un lecteur sur le clavier, même à partir d'une configuration incohérente. Notons que la configuration obtenue par un *reset* ne vérifiera pas toutes les conditions de cohérence, puisque rien n'empêche des références extèrieures aux modules de subsister.

L'exemple de la machine d'entrée de Cds aurait pu être programmé avec des méthodes classiques (il l'était dans les premières versions du système), mais l'implantation en Ceyx est beaucoup naturelle, beaucoup plus lisible, et plus facile à mettre au point. Pour ECRINS, par contre, le gain en puissance et en souplesse est considérable. L'investissement nécessaire à une programmation "classique" d'une machine dynamique aurait été trop important (pour un système expérimental) et n'aurait certainement pas permis autant de souplesse. Notre solution permet un enrichissement futur de l'architecture, même de manière interactive, à un coût très faible (sans modification des structures existantes). Par exemple, nous sommes en train d'implanter en Ceyx les analyseurs lexico-syntaxiques de Syntax [Boullier83a], destinés à remplacer les modules *CxYacc* et *lex-kit* [Berry84b]. Ces modules sont compatibles au niveau du type des objets Ceyx (ils ont des types compatibles, et répondent aux mêmes messages), donc pourront être remplacés sans autre changement. Par contre, ils demandent des fonctionnalités nouvelles au niveau de la machine d'entrée, et l'on peut imaginer, en phase de test, remplacer un analyseur lexicographique fabriqué avec lex-kit par son homologue de Syntax, ajouter dynamiquement les modules nécessaires à la machine d'entrée (source, manager), et tester immédiatement...

## 5 CONCLUSION

Il est très naturel de concevoir les programmes sous la forme d'un réseau de machines virtuelles. Le langage Ceyx est un outil très adapté à la construction d'un tel réseau. Les machines virtuelles trouvent en effet une implémentation immédiate sous la forme d'objets, la communication inter-machines est réalisée par transmission de messages. Ceyx est donc un outil très commode pour construire rapidement des maquettes de systèmes expérimentaux.

Dès que l'on désire construire des systèmes plus "industriels" l'utilisation poussée de Ceyx peut cependant conduire à un certain manque d'efficacité. Les machines d'entrées sont par exemple très modularisées et perdent beaucoup de temps en communication. Dans le cas d'une architecture statique comme celle

de CDS on peut concevoir une compilation du réseau de machines. Une compilation au niveau Lisp consiste à transformer les transmissions de messages inter-modules en appels fonctionnels directs. Nous pensons atteindre sans aucun problème des vitesses comparables à celles obtenues par des programmes proposant les mêmes fonctionnalités mais écrits en Lisp "pur". Une autre compilation consiste à décrire certaines machines du réseau dans le langage ESTEREL (langage de synchronisation temps réel) [Berry84a]. Ces machines peuvent alors être compilées sous forme d'automates dont les transitions sont guidées par les messages reçus des autres machines du réseau. Le formalisme d'ESTEREL reste très proche des machines virtuelles, mais (et même pour une implantation séquentielle) nous attendons des performances bien meilleures. Dans le cas d'une architecture dynamique, comme celle d'ECRINS, la compilation ne peut être que locale (on compile les machines mais non leurs relations dans le réseau). La relative inefficacité inhérente à cette architecture est largement compensée par la souplesse et la puissance de la construction.

## Références

### References


[Austry83a] D. AUSTRY, "Aspects syntaxiques du langage MEIJE," Thèse de troisième cycle, Université de Paris-7 (D).

[Berry83a] G. BERRY AND P.L. CURIEN, "Theory and Practice of Sequential Algorithms : The Kernel of the Applicative Language CDS," Rapport RR 225, INRIA, Rocquencourt (D).

[Berry84a] G. BERRY AND L. COSSERAT, "The ESTEREL Synchronous Programming Language and its Mathematical Semantics," ENSMP (D).

[Berry84b] G. BERRY AND B. SERLET, "Cxyacc et Lex-kit," Rapport interne de l'école des Mines de Paris, Sophia-Antipolis (D).

[Boullier83a] P. BOULLIER, "Contribution à la construction automatique d'analyseurs lexicaux et syntaxiques," Thèse d'état, Université d'Orléans (D).

[Durieux81a] J-L. DURIEUX, *Présentation du langage Plasma, acteurs et continuations*, GRECO de programmation, Université de Bordeaux 1 (1981).

[Hullot84a] J-M. HULLOT, "Programmer en CEYX," Rapport RR..., INRIA (D).