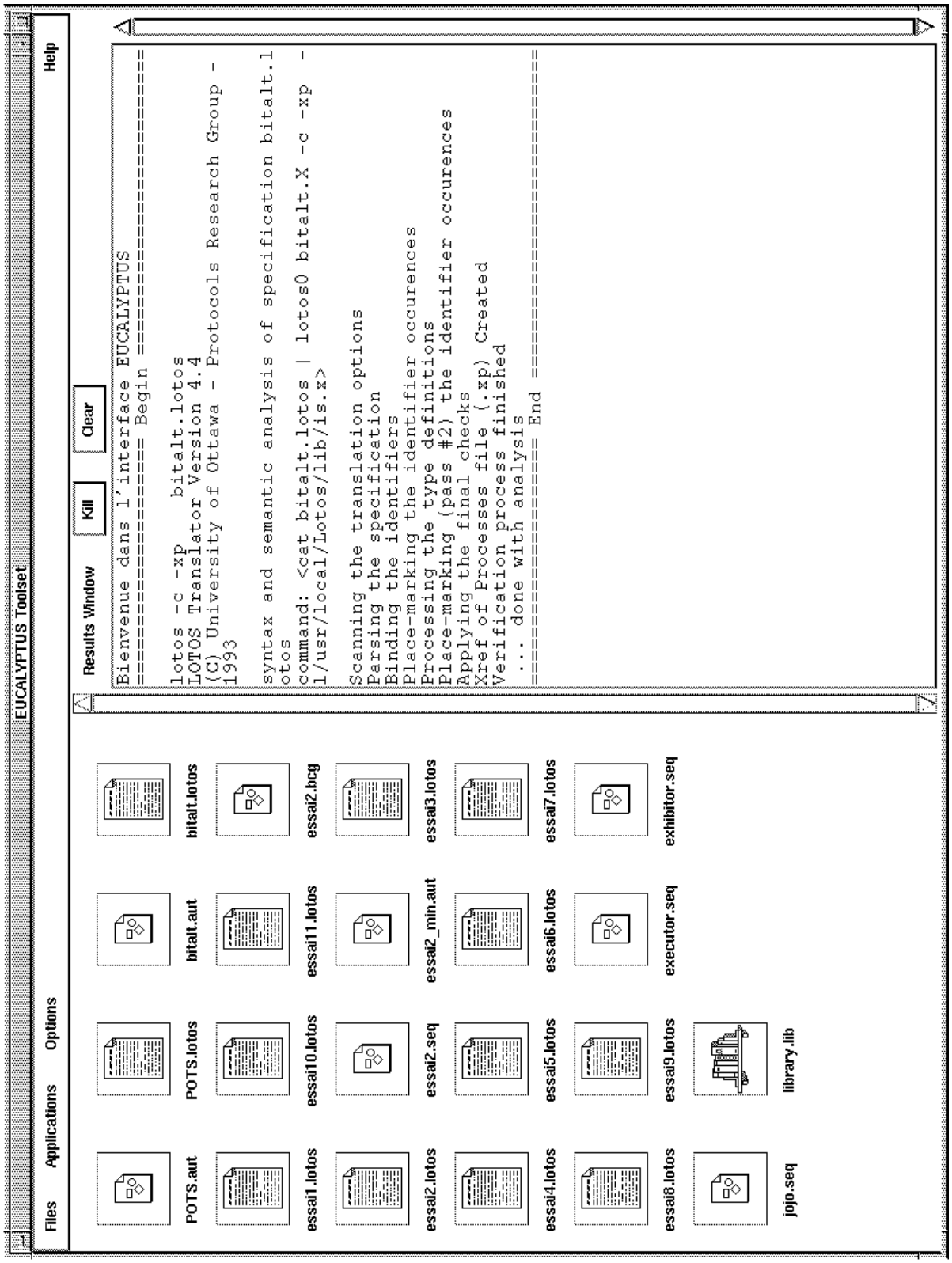
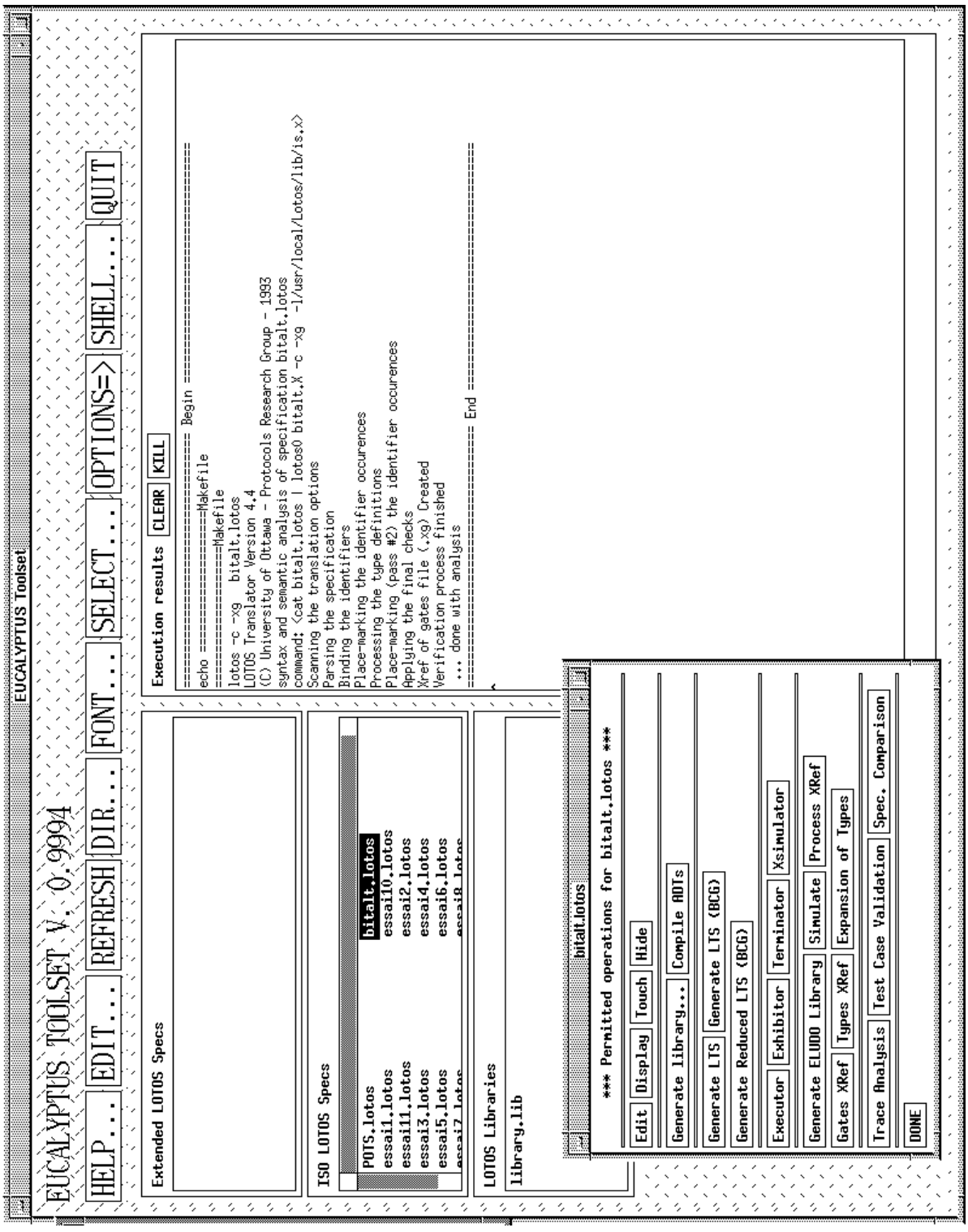
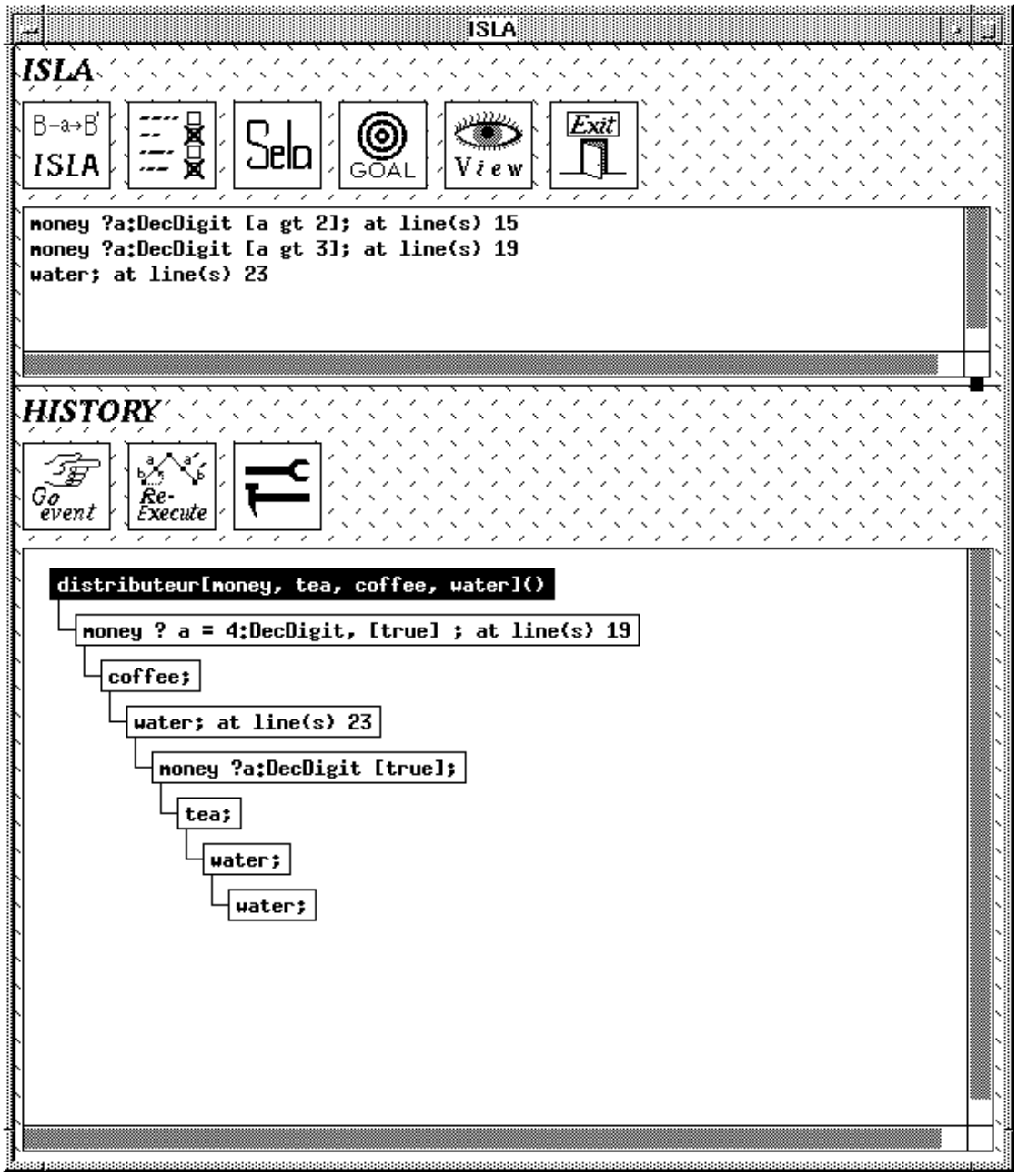
IX.4. La nouvelle Interface Homme-Machine



IX.3. L'ancienne Interface Homme-Machine



IX.2. Exemple d'utilisation de la passerelle :



La sequence qui est alors trouvée est :

*** sequence number 1 found at depth 20

<initial state>
"SUCC2 !CLAIM !STATION2 !ALTERN0"
"PRED1 !CLAIM !STATION2 !ALTERN0"
"SUCC1 !CLAIM !STATION2 !ALTERN0"
"PRED2 !CLAIM !STATION2 !ALTERN0"
"SUCC2 !TOKEN"
"PRED1 !TOKEN"
"OPEN !STATION1"
"CLOSE !STATION1"
"SUCC1 !TOKEN"
"PRED2 !TOKEN"
"OPEN !STATION2"
"CLOSE !STATION2"
"SUCC2 !TOKEN"
"PRED1 !TOKEN"
"OPEN !STATION1"
"CLOSE !STATION1"
"SUCC1 !TOKEN"
"PRED2 !TOKEN"
"OPEN !STATION2"
"CLOSE !STATION2"
<goal state>

```
                        []
                    [ stat_r eq stat ] -> (* on va regarder si la station doit reemettre le jeton *)
                        (
                         [alternate_r eq alternate] -> succ!token;
                        station [pred,succ,open,close](i0,stat,succ(alternate))
                         []

                         [not(alternate_r eq alternate)] ->
                        station [pred,succ,open,close](i1,stat,alternate) (***** voir si eligible ou non ******)
                        )
            )

    )
)
endproc (* station *)

process buffer [succ,pred] : noexit
:=
     ( ( succ?token : token_T;
         pred!token; buffer[succ,pred]
             []
       succ?token : token_T; buffer[succ,pred]  )
         []
     ( succ!claim?stat : adresse?alternate :  alternate_T ;
         pred!claim!stat!alternate; buffer[succ,pred]
             []
       succ!claim?stat : adresse?alternate : alternate_T ; buffer[succ,pred] ) )
endproc (* buffer *)

endspec (* token_ring *)
```

La séquence qui était recherchée avec l'ancienne version d'Exhibitor était :

OPEN !STATION1
OPEN !STATION2
CLOSE !STATION1
CLOSE !STATION2

La séquence recherchée avec le nouvel Exhibitor est :

~ [OPEN.*] *
"OPEN !STATION1"
~ [OPEN.*] *
"OPEN !STATION2"
"CLOSE !STATION2"
~ [CLOSE.*] *
"CLOSE !STATION1"
~ [CLOSE.*] *
"CLOSE !STATION2"

```
  altern0(*! constructor *): -> alternate_T
  altern1(*! constructor *): -> alternate_T
  _eq_ , _ne_ : alternate_T , alternate_T -> Bool
eqns forall
  x_1 , y_1 : alternate_T
ofsort Bool
  altern0 of alternate_T eq altern0 of alternate_T = true ;
  altern1 of alternate_T eq altern1 of alternate_T = true ;
  x_1 eq y_1 = not ( true ) ;
  x_1 ne y_1 = not ( x_1 eq y_1 ) ;
endtype

type alternate_with_succ is alternate_T
 opns
   succ : alternate_T -> alternate_T
 eqns ofsort alternate_T
        succ(altern1)=altern0;
        succ(altern0)=altern1;
endtype

library
Boolean
endlib

behaviour

  ( station[pred1,succ1,open,close](i0,station1,altern0)
        |[succ1]|
    buffer[succ1,pred2] )
        |[pred1,pred2]|
  ( station[pred2,succ2,open,close](i0,station2,altern0)
        |[succ2]|
     buffer[succ2,pred1] )

where

process station[pred,succ,open,close](id : init, stat : adresse,alternate :  alternate_T):noexit :=

(
        succ!claim!stat!alternate; station [pred,succ,open,close](i1,stat,alternate)
                []
        pred?token : token_T;
                ( succ!token; station [pred,succ,open,close](i0,stat,succ(alternate))
                        []
                  open!stat; close!stat;
                  succ!token; station [pred,succ,open,close](i0,stat,succ(alternate)) )
                []
        pred!claim?stat_r: adresse ?alternate_r:  alternate_T ;
                ( [ id eq i0 ] -> succ!claim!stat_r!alternate_r;
                                  station [pred,succ,open,close](id,stat,alternate) (* cas non eligible *)
                                  []

                  [ id eq i1 ] -> (* cas eligible *)
                        ( [ stat_r lt stat ] -> (* cas ou l'on ne fait que passer le claim recu *)
                                  succ!claim!stat_r!alternate_r;
                              station [pred,succ,open,close](i0,stat,alternate) (* non a l'envoie de claim *)
                                  []
                          [ stat_r gt stat ] -> (* cas ou l'on detruit le claim recu et reste eligible *)
                                  station [pred,succ,open,close](i1,stat,alternate)
```

La spécification en question était la suivante:

```
specification token_ring[open, close,pred1, succ1, pred2, succ2] : noexit

type token_T is
sorts token_T
opns
  token(*! constructor *): -> token_T
endtype

type claim_T is
sorts claim_T
opns
  claim(*! constructor *): -> claim_T
endtype

type init is Boolean
sorts init
opns
  i0(*! constructor *): -> init
  i1(*! constructor *): -> init
  _eq_ , _ne_ : init , init -> Bool
eqns forall
  x_1 , y_1 : init
ofsort Bool
  i0 of init eq i0 of init = true ;
  i1 of init eq i1 of init = true ;
  x_1 eq y_1 = not ( true ) ;
  x_1 ne y_1 = not ( x_1 eq y_1 ) ;
endtype

type adresse is Boolean
sorts adresse
opns
  station1(*! constructor *): -> adresse
  station2(*! constructor *): -> adresse
  _eq_ , _ne_ : adresse , adresse -> Bool
  _lt_ , _le_ , _gt_ , _ge_ : adresse , adresse -> Bool
eqns forall
  x_1 , y_1 : adresse
ofsort Bool
  station1 of adresse eq station1 of adresse = true ;
  station2 of adresse eq station2 of adresse = true ;
  x_1 eq y_1 = not ( true ) ;
  x_1 ne y_1 = not ( x_1 eq y_1 ) ;
ofsort Bool
  x_1 lt station1 = not ( true ) ;
  station1 lt y_1 = true ;
  x_1 lt station2 = not ( true ) ;
  x_1 le y_1 = not ( y_1 lt x_1 ) ;
  x_1 gt y_1 = y_1 lt x_1 ;
  x_1 ge y_1 = not ( x_1 lt y_1 ) ;
endtype

type alternate_T is Boolean
sorts alternate_T
opns
```

IX.1.2. Test 2

Le deuxième test consistait à prendre des spécifications Lotos de la forme:

a; exit
|||
a; a; exit
|||
...
|||
a; a; ...; a; exit

parametrées par un entier N et à rechercher la séquence:

- exit (avec l'option -only) avec l'ancienne version
- (~ exit) * . exit . avec la nouvelle version

Résultats:

N	ancienne	nouvelle	états	arcs
1	0	1	3	2
2	0	0	7	8
3	0	0	25	47
4	1	0	121	327
5	3	3	721	2557
6	25	29	5041	22213
7	234	272	40321	212977
8	NC	NC	NC	NC

N : désigne la valeur du paramètre N
ancienne : désigne le temps d'exécution de l'ancienne version
nouvelle : désigne le temps d'exécution de la nouvelle version
etats : désigne le nombre d'états de l'automate
arcs : désigne le nombre d'arcs dans l'automate

Remarque: NC signifie "non calculé", du fait d'un temps d'exécution jugé excessif

IX.1.3. Test 3

Le troisième test consistait à rechercher une séquence donnée dans une spécification de processus ayant accès à une ressource en exclusion mutuelle.

L'ancienne version d'Exhibitor trouvait en 15 secondes environ une trace comprenant 253 labels. La nouvelle version d'Exhibitor trouve maintenant la plus coutre séquence, comprenant 20 labels, en 75 secondes environ.

IX. Annexes

IX.1. Tests d'exhibitor

IX.1.1. Test 1:

Le premier test consistait à prendre des spécifications Lotos de la forme:

```
i; exit
|||
i; i; exit
|||
...
|||
i; i; ...; i; exit
```

paramétrées par un entier N et à rechercher la séquence:

exit

avec l'ancienne et la nouvelle version d'Exhibitor.

Résultats:

N	ancienne	nouvelle	états	arcs
1	0	0	3	2
2	0	1	7	8
3	0	0	25	47
4	1	0	121	327
5	1	1	721	2557
6	4	7	5041	22213
7	36	71	40321	212977
8	421	NC	NC	NC
9	NC	NC	NC	NC

Légende:

N : désigne la valeur du paramètre N
ancienne : désigne le temps d'exécution de l'ancienne version
nouvelle : désigne le temps d'exécution de la nouvelle version
états : désigne le nombre d'états de l'automate
arcs : désigne le nombre d'arcs dans l'automate

Remarque : NC signifie "non calculé", du fait d'un temps d'exécution jugé excessif

VIII. Conclusion

En conclusion, les objectifs que nous nous étions fixés lors de la présoutenance ont été atteints.

En effet, concernant le pont entre CAESAR et ELUDO nous avons proposé trois passerelles distinctes qui fonctionnent dès à présent.

Les extensions apportées au format .seq ont été intégrées dans Exhibitor et dans Bridge.

D'autre part, un algorithme itératif permettant de trouver une plus courte séquence a été implanté dans Exhibitor. Cependant, il semble ne pas donner entière satisfaction d'un point de vue temps d'exécution.

Concernant les outils graphiques, aussi bien l'interface homme-machine d'EUCALYPTUS que l'outil X-Simulator d'OPEN/CAESAR ont été re-écrits en Tcl/Tk.

La nouvelle IHM présente l'avantage d'être plus performante, plus conviviale et il sera dorénavant plus facile de la maintenir et de la faire évoluer.

Cependant, quelques perfectionnements pourraient être envisagés:

- la génération de DDP dans l'outil Goal d'ELUDO
- la prise en compte de l'ensemble du format .seq étendu par Bridge
- l'amélioration de l'évaluateur d'expressions (narrower) d'Eludo
- une amélioration du temps d'exécution d'Exhibitor

Enfin, après la réalisation de ce projet, il serait extrêmement intéressant de poursuivre l'idée d'une coopération plus grande entre les outils de CAESAR et ceux d'ELUDO. En effet, durant ces trois mois, nous avons pu constater que la puissance de calcul de CAESAR associée à la précision d'analyse d'ELUDO pouvaient offrir la meilleure solution, dans la résolution de certains problèmes.

Pour notre part, nous avons apprécié la nouvelle approche du langage LOTOS, c'est à dire, une approche non pas d'utilisateur, mais de "compilateur". Il fut très intéressant de voir tous les mécanismes qui sont mis en oeuvre derrière ce langage.

Nous avons aussi beaucoup apprécié le fait de pouvoir faire notre projet de fin d'études au Canada, et espérons que l'ENSIMAG encouragera ces échanges dans le futur.

Nous allons maintenant étudier comment les listes sont lues par les programmes que nous avons développés.

Par exemple, pour décrire un bouton, nous utilisons cette liste:

```
{Bouton          "Execute CAESAR"          "caesar file.lotos"}
```

Le premier élément de la liste est le type d'objet graphique qu'elle décrit. Les autres éléments lui indiquent les caractéristiques de cet objet. Dans notre cas, le deuxième élément ("Execute CAESAR") indique le label du bouton (ce qui écrit dessus lors de l'exécution du programme) et le troisième élément ("caesar file.lotos") désigne la commande Tcl qui est exécutée lorsqu'un utilisateur clique sur le bouton avec la souris.

Il faut stocker cette liste dans une variable. Pour cela, nous utilisons la commande Tcl **set**.

```
set List {Bouton "Execute CAESAR" "caesar file.lotos"}
```

Notre programme utilise ensuite cette variable pour dessiner ce bouton. Il exécute cette suite de commandes:

```
if Premier_Element($List) = "Bouton"
then
    button nom_interne -label Deuxième_Element($List)
                       -command Troisième_Element($List)
end if
```

La commande "button" est fournie par Tk. Elle permet de définir un bouton. Les caractéristiques de ce bouton lui sont passées en paramètres. Un nom interne d'objet généré par notre programme lui est également transmis.

Si les caractéristiques du bouton changent, il suffit de modifier la liste sans connaître les commandes que le programme exécute réellement.

Les outils d'EUCALYPTUS utilisent de tels fichiers en entrée.

Exemples:

-APERO traduit une spécification LOTOS au format étendu en spécification LOTOS au format ISO.
-CAESAR crée un système de transitions étiquetées à partir d'une spécification LOTOS au format ISO.

Exemple d'action appelée par l'interface:

Pour les spécifications LOTOS au format étendu, il existe une action "Générer Système de Transitions Etiquetées". Lorsque cette action est sélectionnée, l'interface appelle APERO sur cette spécification puis appelle CAESAR sur la spécification LOTOS (au format ISO) générée.

VII.2.3. Caractéristiques internes de l'interface

Elle est constituée de scripts Tcl/Tk et d'un programme C qui permet l'affichage des résultats des commandes sur la fenêtre de droite. Elle a été conçue de façon modulaire. Les scripts Tcl/Tk sont divisés en trois parties:

a) Les fichiers qui décrivent les objets graphiques:

- **DescMenus.tcl** décrit le menu principal.
- **DescOptions.tcl** décrit le menu associé aux options des outils.
- **DescFiles.tcl** décrit les menus associés aux fichiers.

b) Les fichiers associés aux outils EUCALYPTUS: un fichier par outil

c) Les programmes eux-mêmes:

- Les programmes qui créent les objets graphiques:

  - **MakeMenus.tcl** contient les fonctions qui créent les menus.
  - **MakeIcons.tcl** contient les fonctions qui créent les icônes des fichiers.
  - **MakeWindows.tc**l contient les fonctions qui créent les fenêtres.

- Les autres programmes:

  - **MainWindow.tcl** dessine la fenêtre principale.
  - **ExecuteTool.tcl** permet de lancer les commandes associées aux outils.
  - **Communication.tcl** gère l'écriture des résultats des outils.
  - **Browser.tcl** permet l'utilisation de sélecteurs de fichiers.
  - **Help.tcl** offre de l'aide.

Les caractéristiques de Tk

Tk signifie ***Toolkit for X-Window*** (boîte à outils pour X-Window). Sans Tk, Tcl n'aurait aucun intérêt. Tk fournit un ensemble de procédures Tcl afin de permettre la création d' objets graphiques (ou widgets). Un objet graphique est caractérisé par son apparence et son comportement. Il existe plusieurs types d'objets: fenêtres, boutons, menus, éditeurs de textes et de dessins...

Tk gère ces objets sous une forme hiérarchique fournie par le système X-Window. Une interface graphique a toujours une fenêtre principale qui contient elle-même plusieurs objets (appelés fils). Chaque objet peut ainsi contenir d'autres fils. Le nom d'un objet graphique indique sa position dans cette hiérarchie: il comprend les noms de tous ses pères séparés par des points.

Des gestionnaires de positionnements contrôlent la taille et l'emplacement des objets à l'écran. Il existe deux gestionnaires différents: le *packer* (placement selon des mot-clefs) et le *placer* (placement selon des coordonnées).

Enfin, Tk connaît les évènements X-Window liés au clavier et à la souris. Il permet de définir des associations entre de tels évènements et des scripts Tcl. Ces scripts s'exécutent seulement si ces évènements ont lieu.

VII.2.2. Aspect extérieur

L'interface a un aspect extérieur plus attrayant. Elle contient des objets graphiques plus agréables à utiliser.

- Les boutons situés en haut de l'écran ont été remplacés par des menus, de type Microsoft-Windows pour PC.

- La fenêtre de gauche contient des fichiers représentés par des icônes sur un fond de nuages. A chaque type de fichier correspond une icône différente. Ces fichiers peuvent être triés par nom ou par type. Il est possible de choisir les types de fichiers affichés. Pour exécuter une action sur un fichier, un utilisateur doit cliquer sur son icône. Les actions possibles sur ce fichier s'affichent alors sous la forme d'un menu.

- La fenêtre de droite contient les résultats des commandes, comme dans l'ancienne interface.

Les types de fichiers connus sont les mêmes que ceux de l'ancienne interface. Les principaux sont:

- Spécifications LOTOS au format ISO
- Spécifications LOTOS au format étendu
- Systèmes de transitions étiquetées
- Fichiers au format Prolog utilisés par Eludo

développement avec un outil plus performant que Xtpanel: Tcl/Tk. Ainsi, la nouvelle interface devait être plus modulaire, portable et avoir un aspect extérieur plus attrayant.

VII.2.1. L'outil de développement Tcl/Tk

Tcl/Tk a été écrit par John Ousterhout. C'est un outil qui permet de développer des applications graphiques sous l'environnement X-Window. Son principal atout est que son utilisation ne nécessite aucune connaissance sur les bibliothèques du système X-Window. Il est composé de deux parties:

Le langage Tcl

Tcl signifie ***Tool Command Language*** (langage pour commander les outils). C'est un langage, généralement interprété, comparable aux shells UNIX. Sa syntaxe est relativement simple, ce qui le rend facile à apprendre. Les données qu'il manipule sont des chaînes de caractères. Un type de données un peu plus évolué est également disponible: les listes de chaînes. Ces données peuvent être stockées dans des variables ou des tableaux de variables.

Il existe des commandes pour manipuler les chaînes de caractères et les listes. La syntaxe de l'appel d'une commande est simple:

```
command arg1 arg2 ... argN
```

Ainsi un programme (ou script) Tcl est une suite de commandes et de définitions de commandes. De plus, il est possible d'utiliser des structures de programmes qui existent dans les langages de programmation procéduraux:

```
if condition then command1 else command2
while condtion command
etc.
```

La notion de substitution est une caractéristique importante de Tcl. Il existe deux types de substitution:

- la substitution d'une variable: si dans un script, un nom de variable est précédé par un $, il est remplacé par la valeur de la variable.

- la substitution d'une commande: si une commande est délimitée par des crochets, l'interpréteur de Tcl la remplace dans le script par le résultat de cette commande.

Enfin, Tcl a d'autres caractéristiques intéressante.Il permet d'appeler des commandes UNIX et d'effectuer des entrées-sorties sur des fichiers.

VII.1.2. Aspects extérieurs

L'interface se compose de trois éléments:

- à gauche, une fenêtre, qui est divisée en plusieurs catégories, liste les fichiers disponibles dans le répertoire courant: fichiers Lotos, bibliothèques, systèmes de transitions étiquetées, ... En cliquant sur un fichier, on fait apparaître une petite fenêtre, qui présente les opérations permises sur ce fichier.

- à droite, une autre fenêtre présente le résultat de l'exécution de commandes. Un bouton (FLUSH) permet de vider cette fenêtre. Un autre, KILL, permet d'arrêter le processus en cours.

- une rangée de boutons se situe en haut de l'écran:

  - HELP : donne des informations sur les différents outils.
  - EDIT : lance un éditeur sans fichier initial.
  - REFRESH : remet l'écran à jour par rapport aux fichiers se trouvant dans le répertoire courant.
  - DIR : permet de modifier le répertoire courant.
  - FONT : permet de changer la fonte d'affichage des résultats.
  - SELECT : permet de modifier les catégories de fichiers affichés dans la fenêtre de gauche.
  - OPTIONS : permet de configurer les options des outils.
  - SHELL : lance un xterm muni d'un shell UNIX.
  - QUIT : permet de quitter l'interface.

VII.1.3. Les limites

Cette interface connaît un certain nombre de limites dues à la façon dont elle a été conçue et à l'outil de développement XtPanel:

- elle n'a pas été réalisée de façon modulaire, ce qui rend toute modification, même mineure, assez difficile.

- elle n'est pas portable, sur d'autres architectures, sans effectuer d'importantes modifications.

- Xtpanel est très lent et sa gestion des objets graphiques est limitée.

VII.2. La nouvelle IHM

Pour toutes ces raisons, l'Inria Rhône-Alpes (VERIMAG) nous a chargé de réaliser une nouvelle interface homme-machine. Le but était de remplacer l'ancienne en recommençant le

puisque cet solution a l'avantage de faire un maximum de traitements en C, ce qui fait qu'il est plus rapide. Il faut noter aussi que la première solution permet d'utiliser au mieux la puissance de calcul de CAESAR et la précision de résolution d'ELUDO.

VI.3. Troisième solution: Combinaison de Exhibitor et GOAL

GOAL est un outil d'ELUDO qui cherche des buts (un but est une transition) dans une spécification. Il peut aussi lire une liste de buts, et trouver un chemin qui passe par tous ces buts. Nous avons donc ajouté à Exhibitor une option (option goal), afin de générer des traces pouvant être lues par GOAL.

Puisque GOAL fait partie de ELUDO, ceci constitue une communication entre CAESAR et ELUDO. Néanmoins, ce qui nous intéresse dans le passage de CAESAR à ELUDO, c'est de pouvoir rejouer la trace, donc d'utiliser ISLA. Il n'est pas encore possible de passer de GOAL à ISLA, mais nous avons reçu l'assurance que cela sera possible prochainement.

Il s'agit donc d'une passerelle qui sera opérationnelle prochainement, dès que la communication de GOAL à ISLA sera implanté.

VII. L'interface d'EUCALYPTUS

Ce chapitre concerne l'interface homme-machine que nous avons réalisée. Dans un premier temps, nous étudierons le rôle et les limites de l'interface qui existait déjà. Ensuite, nous présenterons l'outil de développement que nous avons utilisé et les caractéristiques de la nouvelle interface

VII.1. L'IHM existante

VII.1.1. Développement

Pour réunir leurs outils dans une boîte à outils cohérente, les partenaires du projet EUCALYPTUS ont développé une interface graphique (GUI pour Graphical User Interface).

En 1993, l'Inria Rhône-Alpes (VERIMAG) a défini les spécifications de l'interface pour les outils CAESAR, CAESAR.ADT, ALDEBARAN et OPEN/CAESAR. Le laboratoire de Liège les a définies pour APERO. L'ensemble des partenaires a choisi le système X-Window (version X11R5) parce que l'interface graphique d'ELUDO était déjà basée sur X-Window.

Sous la proposition de l'Inria, l'outil XtPanel (associé à l'environnement de développement MOTIF) a été choisi. L'Inria a développé un premier prototype pour l'outil ALDEBARAN. Sur cette base, les partenaires ont fixé l'apparence générale de l'interface et le déroulement de sa réalisation, qui a duré de mars 1994 (version 0.5) à décembre 1994 (version 0.998). Chaque laboratoire a effectué l'intégration de ses outils dans l'interface. Le code, résultant de cette activité collective, contient 5.500 lignes en langage XtPanel.

coûteux. En effet, dans le cas où l'utilisateur écrit:

```
"MONEY !1*0"
```

lorsque dans la spécification il y a :

```
"MONEY ?x:nat"
```

Cela ajoute des restrictions sur x que nous ne savons pas vraiment formaliser. Il faudra donc obtenir une valeur avec le narrower, vérifier si celle-ci convient, et si elle ne convient pas, obtenir à nouveau une valeur avec le narrower, et boucler ainsi de suite jusqu'à satisfaction. Ce qui pourra prendre longtemps.

Il n'est donc pas raisonnable de permettre les expressions régulières dans le cas de description de porte avec offres.

Parcours

Nous allons, tout comme Exhibitor, parcourir la spécification pour trouver le chemin correspondant à la trace fournie. Le parcours est le même que celui du programme bridge que nous avons étudié plus haut. Il s'agit donc d'un parcours en profondeur avec du retour-arrière Prolog, s'appuyant sur le prédicat *infer* pour obtenir les transitions suivants, et produire le DDP en route.

Plusieurs problèmes ont été rencontrés:

- ELUDO traduit la spécification en une base de données Prolog. Nous ne parcourons pas un automate, mais une spécification. Le problème qui se pose alors est celui de repérer ce qui correspond aux états, c'est à dire les comportements, pour pouvoir détecter un cycle éventuel, pour détecter que nous sommes déjà passés par cet état. Pour résoudre ce problème, il est nécessaire d'effectuer un "nettoyage" des comportements, pour pouvoir par la suite les comparer.

- Comment éviter de boucler lors du parcours, et cela notamment lorsque l'utilisateur utilise les fonctions de redondance "*" dans la grammaire qu'il fournit.

Pour cela, il faut détecter si nous visitons un état qui a déjà été visité, si nous avons avancé dans notre grammaire, si nous pouvons prendre une transition différente que celle qui a été prise lors de la dernière visite. Dans l'implémentation, nous créons une liste qui décrit les états déjà visités, le numéro de règle dans la grammaire qui fut utilisé pour nous amener dans cet état, ainsi que les transitions qui furent prises lors des dernières visites.

- Comment obtenir les valeurs des offres, lorsqu'elles ne sont précisées dans la trace ?

Ce problème fut aussi rencontré dans l'implémentation du programme Bridge, vu plus haut. Il se résout ici, exactement de la même manière: En utilisant le narrower.

Résultats

Le programme produit des résultats assez satisfaisants. Il trouve par exemple les chemins correspondant à une trace dans le POTS en un temps "assez raisonnable". Mais d'une manière générale, ce programme est très lent. Le fait qu'il soit écrit en Prolog ajoute à sa lenteur. Nous recommandons donc plutôt la première solution pour communiquer entre CAESAR et ISLA,

batterie de tests très complète de 56 automates différents et intéressants fut élaboré.

Le module *automate* est surtout un module d'intégration des programmes dans l'environnement. Intégration dans ELUDO pour bridge, et intégration dans CAESAR pour Exhibitor. Nous avons donc surtout effectué des tests d'intégration.

Nous avons voulu voir comment réagissait Bridge, lorsque les spécifications sont très grandes, et que les traces à intégrer sont aussi très longues. Ainsi, nous avons cherché une séquence de longueur 100 dans la spécification du système téléphonique (POTS: Plain Old Telephone System). Bridge a produit un fichier checkpoint, qui a pu être rejoué par ISLA. Pour produire ce fichier, il a fallu 5'24'' de temps CPU, sur un Sun ULTRA.

VI.1.9. Exemple d'utilisation de cette passerelle

Nous avons vu dans le paragraphe V.4. un exemple d'utilisation d'Exhibitor. Nous avons pris le résultat obtenu dans cette exemple, pour l'injecter dans Bridge. Nous avons alors obtenu un fichier Checkpoint que nous avons rechargé avec ISLA. Nous voyons alors tout le chemin qui a été parcouru, et nous pouvons rejouer cette trace, et faire par exemple des choix différents à chaque étape. La fenêtre d'ISLA où la re-exécution se fait, pour cet exemple bien précis, peut être trouvé dans l'annexe IX.3 .

VI.2. Deuxième solution: Production de "Checkpoint" à partir de ".seq étendu"

Nous avons développé un programme Prolog, ayant exactement la même architecture que le programme *Bridge* vu précédemment. Celui-ci prend en entrée un fichier ".seq" étendu, dont le format est décrit dans le chapitre IV. Il utilise les mêmes fonctions de correspondance qu'Exhibitor, qui sont décrits dans le paragraphe V.3.

Ce programme s'appuie directement sur les différents modules du noyau ELUDO.

Restrictions

Néanmoins, les fichiers .seq étendus qu'il prend en entrée ne doivent pas avoir d'expressions régulières dans la description de transitions sur des Portes avec offres.

Par exemple,

```
"MONEY !*"
```

est interdit. Mais,

```
"Te*"
```

est permis, dans le cas où il n'y a pas d'offres attendus. Donc, dans ce cas le symbole "*" ne représente que la fin du nom de la porte, ici, Tea.

Il y a une raison assez simple qui justifie cette restriction. Il est très coûteux de fabriquer les valeurs des offres que la trace ne fournit pas. Comme nous avons vu dans la partie précédente, nous devons faire appel au *narrower* et celui-ci est assez lent. Si de plus l'utilisateur ajoute des restrictions "textuelles" sur les résultats obtenus par le narrower, cela va être de plus en plus

interface.pl : Il s'agit d'une interface entre des fonctions écrits en C et les programmes Prolog qui les appellent. La fonction C qui est dans ce cas, est le parseur de chaînes C.

corresp.pl : Ici le prédicat *transition_suivante* trouve, à partir du chemin parcouru jusqu'à présent, de la trace, et de l'ensemble des transitions possibles, les transitions qui sont acceptables.

automate.pl : Ici, le prédicat *quelle_transition* trouve, à partir du comportement courant, l'ensemble des actions possibles, les comportements dérivés, les DDP associés aux transitions, et la liste des gardes associées. Il appelle le prédicat *infer* décrit plus haut, plusieurs fois, pour obtenir toutes les transitions possibles, à partir d'un comportement donné.

parcours.pl : C'est ici que nous effectuons le parcours de la spécification. Il s'agit d'un parcours en profondeur d'abord avec retour arrière Prolog.

bridge.pl : C'est ici que nous initialisons les différents modules, que nous chargeons en mémoire la spécification LOTOS, ainsi que la trace, qui pour des raisons historiques s'appelle grammaire. Ce module comporte un *runtime_entry* qui fait que nous obtenons un exécutable *bridge.exe*. Celui-ci interagit avec l'utilisateur pour obtenir le nom des fichiers entrée. Ceux-ci doivent être entrés par l'utilisateur en respectant les normes Prolog : le nom doit être écrit entre apostrophes, et il faut mettre un point à la fin de la ligne.

VI.1.8. Tests

Les remarques suivantes sont valables aussi bien pour le développement d'Exhibitor que pour celui de Bridge.

Le premier module à avoir été écrit fut le module *syntax*. Il s'agissait d'un analyseur syntaxique. Il fut testé exhaustivement, tout seul. C'est à dire que c'était uniquement ce module qui était testé. Des tests constructifs et destructifs furent conçus. Toutes les règles de la grammaire furent couvertes par les tests. En ce qui concerne les tests destructifs, notre philosophie fut de prendre chaque élément de la grammaire, et de concevoir un test où cet élément précis a une anomalie.

Une fois que le module *syntax* fonctionnait correctement, le module *corresp* fut développé. Il s'agissait essentiellement, de reconnaître des phrases appartenant à la grammaire fournie par l'utilisateur. Ce module s'appuie sur le module *syntax.* Donc il n'y a pas eu besoin d'écrire un stub pour tester *corresp*.

Ensuite, nous avons implanté le module *parcours.* Pour pouvoir le tester, un stub fut écrit pour remplacer le module *automate.* Ce stub simulait un automate, qui était décrit par le testeur par un fichier au format ".aut" soit un des formats utilisés par les outils de CAESAR et donc permettait au testeur de visualiser graphiquement ces automates avec l'utilitaire *draw_2D.* Une

parcouru a une très forte probabilité d'être valide, ce qui fait que l'évaluation à la fin des gardes a de très grandes chances d'aboutir.

Les facteurs qui font que le chemin parcouru, sans avoir évalué toutes les gardes, a de très grandes chances d'être valide sont:

- La trace fournie est une trace valide.
- Elle est très détaillée: Les portes invisibles sont précisées.
- Elle est complète: il ne manque aucune transition.

En outre, nous sommes convaincus que dans une spécification correspondant à une application réelle, il n'existe que très peu de chemins dans la spécification qui correspondent exactement à la trace fournie.

Nous pourrions alors nous demander pourquoi nous faisons quand même l'évaluation des gardes manquantes, alors que le chemin associé a de très fortes chances d'être valide.

La cause de cela est assez simple. L'évaluation des gardes par le narrower permet de donner une valeur correcte aux variables non instantiés. Or il faut que toutes les variables qui interviennent dans les offres LOTOS soient instantiées, si nous voulons rejouer la trace. En effet, plus haut, nous avons vu que pour produire un fichier "checkpoint", il faut deux choses: la liste des DDP ainsi que la liste des expériences. Les expériences, comme nous allons voir dans le paragraphe suivant, sont les valeurs que prennent les variables LOTOS lors des offres.

VI.1.6. Les Expériences

Il s'agit d'une liste associant une valeur à une variable, valeur que prend la variable lorsque celle-ci est liée par une offre. Cette valeur peut être produite par le narrower lors de l'évaluation des gardes, ou fournie par la trace. Dans le cas où la variable n'a pas de valeur après l'évaluation des gardes, et que sa valeur n'a pas été fournie par la trace, c'est que sa valeur n'est pas très importante, et qu'elle prendre n'importe quelle valeur de la bonne sorte. Nous prenons alors pour valeur n'importe quelle valeur atomique de la sorte attendue.

Exemple d'expérience:

```
Spec            Money ?x:Bool ;
Trace           Money !true
Expérience      expers(x,s87,true)
```

VI.1.7. Architecture de l'implémentation

Nous avons développé plusieurs modules Prolog:

syntax.pl : Il s'occupe de l'analyse syntaxique des traces. Il vérifie que ceux-ci sont bien écrits au format .seq légèrement modifié (les portes avec offres sont des chaînes C et donc doivent être écrits entre guillemets: "Money !1"), et récupère, quand nécessaire les informations qui se trouvent dans les commentaires.

VI.1.5. Validité du chemin

La condition nécessaire et suffisante pour qu'un chemin (liste de DDP) dans la spécification LOTOS soit valable, est que l'ensemble des gardes et des prédicats associés aux actions s'évalue à vrai.

Cette évaluation est effectuée, quand cela est raisonnable, à la volée, lors du parcours de la spécification. Précisons ce que nous voulons dire par "quand cela est raisonnable" et voyons un cas où cela ne l'est pas.

Il est possible d'évaluer une garde ou un prédicat, lorsque toutes les variables qui se trouvent dans cette garde ou dans ce prédicat, ont été instantiées. Il peut arriver qu'elles ne le soient pas, comme l'illustre l'exemple suivant.

Spécification LOTOS:

```
Hide Money in
    Money ?x : nat ;
    [(x gt 5) and (x lt 8)] -> Tea
```

Trace :

```
i ^a#$% Money ^b
Tea
```

Nous remarquons dans cet exemple que la porte money est une porte cachée. La trace fournie par Exhibitor nous indique que nous nous attendons à une porte invisible. Sur la même ligne, dans le commentaire, Exhibitor nous précise de quelle porte il s'agit. Mais nous ne connaissons pas l'offre, ainsi nous ne pouvons pas donner une valeur à la variable x, et donc nous ne pouvons pas évaluer directement la garde [(x gt 5) and (x lt 8)].

Il est néanmoins possible d'évaluer cette garde en appelant le *narrower.* Il s'agit d'un évaluateur d'expressions, qui donne des valeurs à ce qui est non instantié, pour que la garde puisse être évaluée à vraie. Celui-ci fait parti du noyau ELUDO, et est utilisé surtout par le programme GOAL, qui cherche un but dans une spécification, et doit donc pour cela trouver un chemin dans la spécification et évaluer les gardes associées.

La solution idéale serait que CAESAR puisse préciser non seulement les noms des portes cachées lors des transitions invisibles, mais aussi la valeur des offres. En fait l'implémentation de Bridge permet de prendre en compte la valeur de l'offre, si celle-ci est précisé avec le nom de la porte dans le commentaire, comme ceci:

```
i ^a#$% "Money !6" ^b
```

Cette solution permet de se passer entièrement du narrower, mais malheureusement, CAESAR ne fournit pas encore la valeur des offres.

Cet évaluateur a été écrit en Prolog, et a donc l'inconvénient d'être très lent. Si nous évaluons les gardes à la volée avec le narrower, nous passerons du temps à faire l'évaluation pour des chemins qui, plus tard ne correspondront pas à la trace. Il nous a donc semblé plus raisonnable de retenir les gardes qui n'ont pas été évalués à la volée pour les évaluer à la fin, lorsque le DDP est produit, par le *narrower.* Ceci est d'autant plus raisonnable que le chemin

aboutir au comportement final.

Nous pourrions nous demander pourquoi faut-il un tel code pour décrire la transition entre deux comportements, pourquoi l'action sur laquelle se fait la transition ne suffit pas ? Etudions l'exemple suivant pour mieux appréhender le problème et répondre à cette question.

Soit la spécification LOTOS suivante:

```
B1;
(a !1 ; B2
[]
a !1 ; B3 )
```

Dans l'état actuel de la simulation, nous venons d'effectuer la transition B1 . A présent nous avons donc le choix entre **a !1** et **a !1**. Puisque les deux transitions se font sur la même action, celle-ci ne suffit pas à décrire précisément comment nous passons du comportement courant au comportement suivant. Le seul moyen de décrire précisément la transition est de retenir les règles d'inférence qui ont été utilisées. Ainsi, dans les cas de choix, le numéro de choix est retenu. Dans notre exemple, le mot-clé **ch^1** du DDP correspond à la branche du haut (1er choix), et **ch^2** à la branche du bas (2nd choix).

VI.1.4. Production des DDP

Ces DDP sont produits lors de l'application des règles d'inférence au comportement. Nous allons donc utiliser ce moyen pour produire les DDP. Plus précisément, nous utilisons le prédicat *infer(Beh,Act,DerBeh,DDP,GardeL)* du module *inf.pl* du noyau ELUDO, qui dérive le comportement Beh sur l'action Act pour aboutir au comportement DerBeh. Il produit en route, le DDP associé à la transition, ainsi que la liste des gardes associées à l'action.

Pour produire l'ensemble des DDP décrivant le chemin entier qui correspond à la trace, nous allons parcourir la spécification LOTOS, en allant d'un comportement à un autre en appliquant les règles d'inférence, en choisissant toujours l'action qui correspond à ce qui est dans la trace. Nous appliquons donc l'algorithme suivant:

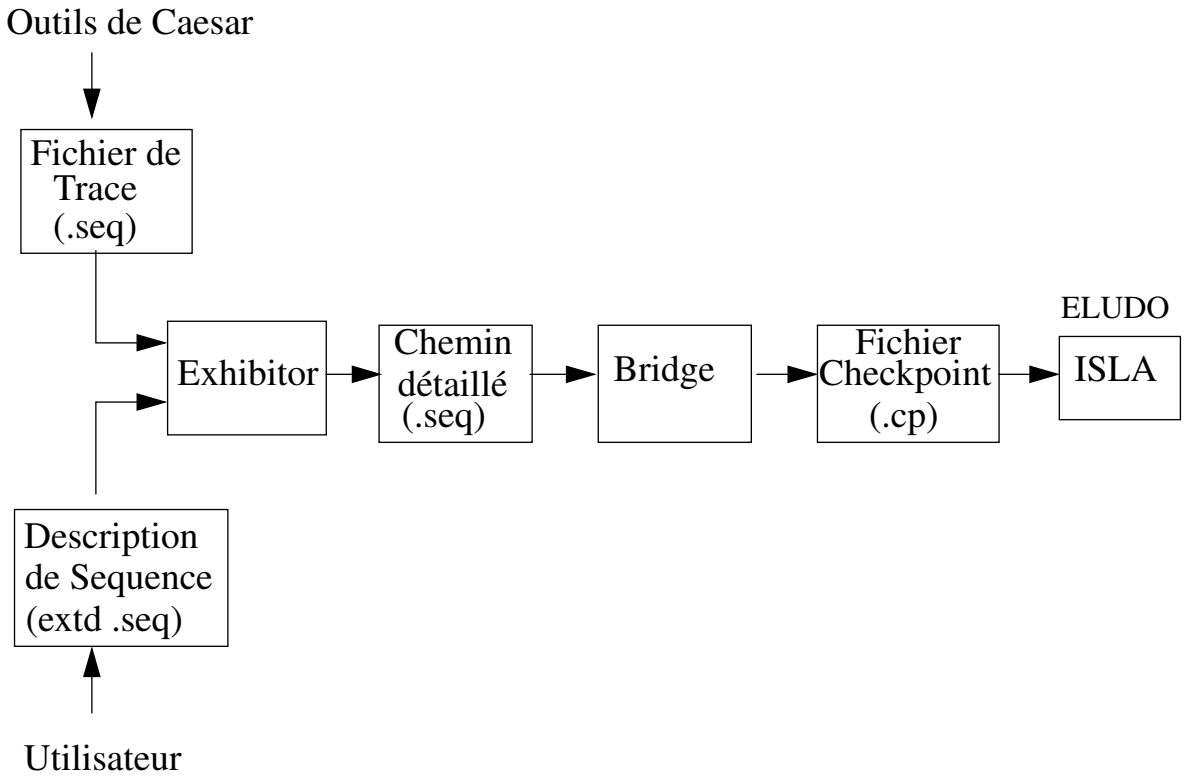
1) appliquer le prédicat infer au comportement courant
2) choisir l'action qui correspond à la trace
3) retourner à 1) avec comportement courant = comportement dérivé, associé à l'action choisie, jusqu'à ce que la liste de trace soit vide.

En cas où nous arrivons à un comportement à partir duquel aucune des prochaines actions possibles ne correspond à ce qui est dans la trace, nous retournons automatiquement en arrière en utilisant les facilités de Prolog.

Il s'agit donc d'un parcours en profondeur de la spécification avec un retour arrière Prolog.

ELUDO, nous nous rapprochons de la spécification LOTOS. Dans la partie précédente, nous avons beaucoup parlé de parcours d'automates. Ici, nous allons parcourir la spécification LOTOS. Les états que nous avions autrefois, correspondent ici à des comportements LOTOS, les transitions sont des actions LOTOS.

VI.1.1. Architecture de cette passerelle



VI.1.2. Production de fichiers "checkpoint"

Il existe un module dans le noyau ELUDO, qui produit de tels fichiers, il s'agit du module *chkpt.pl*. Les prédicats de ce module ont besoin de deux éléments pour pouvoir produire un fichier "checkpoint": - une liste de DDP (Dynamic Derivation Path)
- une liste d'expériences

Voyons ce que sont exactement ces deux éléments et comment ils peuvent être produits à partir de la trace détaillée laissée par Exhibitor et la spécification LOTOS original.

VI.1.3. Les DDP

DDP est l'acronyme de "**Dynamic Derivation Path**", ce qui veut dire, en français, "Chemin de Dérivation Dynamique". Il s'agit d'un code interne au noyau ELUDO qui décrit de manière non ambiguë une transition entre deux comportements LOTOS. Plus précisément, il s'agit d'une liste des règles d'inférence qui ont été appliquées au comportement initial pour

Nous avons implémenté trois moyens pour permettre la communication entre CAESAR et ELUDO:

- une combinaison de Exhibitor avec un programme qui permet de produire des fichiers checkpoint
- produire des fichiers checkpoint à partir des fichiers .seq étendus, directement, par un module complètement intégré à ELUDO.
- produire un fichier au format lisible par l'outil GOAL (qui fait partie de ELUDO), par Exhibitor, en sachant que prochainement il sera possible de passer de GOAL à ISLA.

Nous allons donc étudier dans cette partie une description de ces moyens de communication, notamment de la première, qui est de loin la plus efficace, et voir quelles sont les raisons des choix que nous avons effectués.

VI.1. Intégration des résultats de Exhibitor dans ELUDO

Est-il possible de produire un fichier checkpoint directement, avec Exhibitor ?

Un problème de taille se pose. En effet, ELUDO, qui a une approche beaucoup plus "liée" à la spécification Lotos que CAESAR, demande beaucoup d'informations que ce dernier ne "récolte" pas durant la conversion qu'il fait de la spécification Lotos en un automate. D'autre part, ELUDO ayant été écrit en Prolog, le fichier checkpoint est un pseudo-code Prolog difficile à générer à partir d'un langage procédural de type C.

Nous voyons donc, qu'il est très difficile, voire impossible, de générer un fichier checkpoint dans CAESAR. De plus, en ce qui concerne la maintenabilité de l'interface, il serait pas très sage de produire "manuellement" un fichier checkpoint, alors que ce format peut un jour changer. Il est beaucoup plus raisonnable, de ne pas descendre dans des niveaux de codage aussi bas, et d'utiliser les procédures déjà existantes dans ELUDO, qui permettent de produire de tels fichiers. Nous nous plaçons alors à un niveau d'abstraction plus élevé, ce qui fait que nous appelons des procédures pour produire le fichier checkpoint, mais nous ne savons pas du tout comment ce code est produit.

Nous avons donc décidé que la génération d'un tel fichier se ferait dans ELUDO (plus précisément dans Isla), à partir des résultats de Exhibitor. Pour faciliter la tâche, à ce programme d'intégration, nous avons fait en sorte qu'Exhibitor exploite au mieux les informations qu'il trouvait lors de son parcours de graphe, en les lui transmettant. Notamment, lorsqu'il se produit une transition de type invisible, Exhibitor précise s'il s'agit d'une transition invisible explicite (le i du langage LOTOS) ou s'il s'agit d'une porte cachée, et dans ce cas précise le nom de la porte. La transmission de ces informations se fait dans les commentaires du fichier .seq que produit Exhibitor.

A cet endroit-ci de l'interface logicielle, nous cherchons donc à nous rapprocher le plus possible de ELUDO. Ainsi, les programmes qui réalisent l'intégration des résultats sont écrits en Quintus Prolog, c'est à dire le langage dans lequel les outils de ELUDO ont été écrits. Ils s'appuient sur les prédicats du noyau ELUDO. Au fur et à mesure que nous nous rapprochons de

Nous obtenons le résultat suivant :

```
^A*** looking for sequence of the form:
(i)* . (~(COFFEE))* . (i)* . (COFFEE) . (i)* . (MONEY !0 | ~(MONEY.*))+ . (i)* .
(MONEY !0 | (MONEY !1 | (MONEY !2 | MONEY !3))) . (i)* . (TEA | COFFEE) . (i)* .
(~(WATER))* . (i)* . (WATER) . (i)* . (~((TEA | (COFFEE | WATER))))* . (i)* . (TEA |
(COFFEE | WATER)) . ^B

^A*** sequence number 1 found at depth 7 ^B

^A<initial state>^B
"MONEY !4"
 COFFEE
WATER
"MONEY !3"
TEA
 WATER
WATER
^A<goal state>^B
```

VI. La communication entre CAESAR et ELUDO

Le but du projet est de permettre à ISLA (le simulateur interactif de spécifications LOTOS, qui fait partie de ELUDO) de rejouer une trace laissée par les outils de CAESAR. Il existe une fonctionnalité dans ISLA qui permet de sauvegarder l'état de la simulation en cours, et de la recharger ultérieurement pour pouvoir reprendre la simulation, et se retrouver exactement dans le même état que lors de la sauvegarde. Il nous a semblé judicieux d'utiliser cette fonctionnalité de ISLA pour pouvoir rejouer les traces. Ainsi l'utilisateur qui rejouerait une trace ne saurait même pas que celle-ci provient d'une simulation avec CAESAR, il aurait l'impression de reprendre une simulation qui aurait été faite avec ISLA. Pour plus d'informations sur ce que nous entendons par re-exécution de traces, le lecteur est invité à se référer au paragraphe III.3, intitulé "*La re-exécution de traces*".

La fonctionnalité de ISLA qui charge et sauvegarde des simulations, produit et lit des fichiers "checkpoint" qui ont le format ".cp". Il suffit donc de produire un tel fichier pour pouvoir rejouer la trace.

Jusqu'alors, seul l'outil TETRA était capable, à partir d'un fichier au format .seq, de générer un fichier checkpoint. Le but principal de TETRA n'est pas la production de fichiers checkpoint, mais l'analyse de traces. La production de ces fichiers est un effet de bord de celui-ci. Ce programme présentait encore des défauts de fonctionnement et était plutôt lent. Il était donc indispensable de permettre à CAESAR et ELUDO de communiquer, sans avoir à passer par TETRA.

V.4. Exemple d'utilisation du nouveau Exhibitor

Voyons maintenant un exemple complet d'utilisation de Exhibitor.

Soit la spécification LOTOS suivante :

```
 process distributeur[money,tea,coffee,water]: noexit :=
money ?a:Nat [a gt 2];
tea;
distributeur[money,tea,coffee,water]
 []
money ?a:Nat [a gt 3];
coffee;
distributeur[money,tea,coffee,water]
 []
water;
 distributeur[money,tea,coffee,water]
 endproc
```

Nous cherchons la séquence suivante :

```
 ^A First user just wants a coffee ^B
 ~ COFFEE *
 COFFEE

 ^A Second user wants to drink but has no money ^B
 "MONEY !0" | ~ [MONEY.*] +

 ^A Third user wants a warm drink but doesn't want to pay more than 3 ^B
 "MONEY !0" | "MONEY !1" | "MONEY !2" | "MONEY !3"
 TEA | COFFEE

 ^A Fourth user just wants water ^B
 ~ WATER *
  WATER

 ^A Last user wants to drink something... ^B
 ~ (TEA | COFFEE | WATER) *
 TEA | COFFEE | WATER
```

si L est de la forme {label_elementaire & label} alors
renvoie Corresp_lbl_elem (E, label_elementaire) et Corresp_label (E, label)

si L est de la forme {label_elementaire | label} alors
renvoie Corresp_lbl_elem (E, label_elementaire) ou Corresp_label (E, label)

si L est de la forme {label_elementaire ^ label} alors
renvoie Corresp_lbl_elem (E, label_elementaire) ou exclusif Corresp_label (E, label)

- Corresp_lbl_elem (E, LE)

cette fonction permet de calculer la correspondance entre une étiquette E (chaîne de caractères) et un label élémentaire LE, au sens de la grammaire du format .seq.

si LE est de la forme {chaine_simple} alors
renvoie Corresp_chaine_simple (E, chaine_simple)

si LE est de la forme {"chaine_C"} alors
renvoie Corresp_chaine_c (E, chaine_C)

si LE est de la forme {[expression_reguliere_Unix_non_vide]} alors
renvoie Corresp_expression_reguliere (E, expression_reguliere_Unix_non_vide)

si LE est de la forme {(label)} alors
renvoie Corresp_label (E, label)

si LE est de la forme {~ label_elementaire} alors
renvoie non Corresp_lbl_elem (E, label_elementaire)

- Corresp_chaine_simple (E, CS)

cette fonction permet de calculer la correspondance entre une étiquette E et une chaîne simple CS. Le résultat de cette fonction est VRAI si le strcmp entre E et CS renvoie 0.

- Corresp_chaine_c (E, CC)

cette fonction permet de calculer la correspondance entre une étiquette E et une chaîne C CC. Pour se faire, tous les caractères, constituant les deux chaînes, sont convertis en \xxx, où xxx désigne le code ASCII du caractère (exemple: a donne \097). Le résultat de la fonction correspond à un strcmp entre ces deux chaînes.

- Corresp_expression_reguliere (E, ER)

cette fonction permet de calculer la correspondance entre une étiquette E et une expression régulière Unix non vide ER. Le résultat de la fonction est égal au résultat de re_exec (cf man regex) entre les deux chaînes.

Dans cette sous section, nous allons donc expliciter la sémantique du format .seq, en décrivant la fonction de correspondance.

Toutes les fonctions, dont nous allons parler par la suite, renvoient des valeurs booléennes.

La sémantique décrite ci-dessous, est celle implantée dans Exhibitor par défaut.

- Corresp_sequence (L, S)

cette fonction permet de calculer la correspondance entre une liste de labels L et une séquence S.

```
si S est de la forme {groupe_de_labels} alors
        renvoie Corresp_grp (L, groupe_de_labels, L')

si S est de la forme {groupe_de_labels <retour_chariot> liste_de_labels} alors
        renvoie Corresp_grp (L, groupe_de_labels, L') et Corresp_sequence (L', liste_de_labels)
```

- Corresp_grp (L1, G, L2)

cette fonction permet de calculer la correspondance entre une liste de labels L1 et un groupe de labels G. La liste L2 constitue la sous-listes des labels de L1 qui n'ont pas été utlisés pour le calcul : on a L1 = LG.L2, avec LG pouvant être mis en correspondance "entièrement" avec G.

```
si G est de la forme {label} alors
        L2 = cdr (L1)
        renvoie Corresp_label (car(L1), label)

si G est de la forme {label *} alors
        tant que Corresp_label (car(L1), label) faire
                L1 = cdr (L1)
        L2 = L1
        renvoie VRAI

si G est de la forme {label +} alors
        renvoie Corresp_label (car(L1), label) et Corresp_grp (cdr (L1), label *, L2)
```

- Corresp_label (E, L)

cette fonction permet de calculer la correspondance entre une étiquette E (chaîne de caractères) et un label L, au sens de la grammaire du format .seq.

```
si L est de la forme {label_elementaire} alors
        renvoie Corresp_lbl_elem (E, label_elementaire)
```

Cet algorithme est donc itératif et ne risque donc pas de provoquer une explosion de la pile. D'autre part, il trouve une solution optimale, puisque la première solution trouvée est forcément une séquence de longueur minimale.

V.2. Choix de l'algorithme et implémentation: Le nouvel Exhibitor

Pour Exhibitor, nous avons choisi d'implémenter la solution basée sur un parcours en largeur de l'automate, car cette méthode semblait être mieux adapté au langage utilisé qu'est le langage C. En effet, la solution d'un parcours en profondeur aurait nécessité de gérer tout un mécanisme de retour arrière, ce qui aurait risqué de pénaliser fortement les performances d'Exhibitor.

D'autre part, le parcours en largeur a l'avantage de ne pas nécessiter un parcours complet de l'automate pour déterminer quel est le plus court chemin.

Nous allons maintenant préciser quelques détails techniques, qui ont été mis au point, afin de limiter les calculs redondants:

- table_etats : une table de hachage, contenant des listes de BLOCS, est mise à jour (elle contient l'ensemble des BLOCS déjà rencontrés), afin de ne pas avoir à calculer plusieurs fois une même partie commune entre deux chemins. Ainsi, dans l'automate:

  (0, a, 1)
  (0, b, 2)
  (1, c, 3)
  (2, d, 1)
  (3, e, 4)

  lorsqu'on s'aperçoit que le chemin <(0, b, 2) (2, d, 1)> revient dans l'état 1, où l'on est déjà passé avec le chemin <(0, a, 1)>, on ne poursuit pas la recherche, avec ce chemin, si, bien évidemment, on est dans le même état d'avancement au niveau de la séquence spécifiée par l'utilisateur.

- cycle_inutile : un bloc B sera jugé comme dernier maillon d'un cycle inutile, si, dans le chemin utilisé pour y arriver, on a déjà rencontré B.etat et que l'on était dans le même état d'avancement au niveau de la séquence spécifiée par l'utilisateur.

Des tests de performance ont été réalisés sur l'ancienne et la nouvelle version d'Exhibitor et sont disponibles en annexe.

V.3. La fonction de correspondance

Cette fonction constitue le "coeur" sémantique, associé au format .seq. En effet, elle permet de savoir si une liste de labels (étiquettes associées aux arcs d'un LTS) notée L, peut constituer un chemin et donc être mise en correspondance avec une séquence S, au format .seq, fournie par l'utilisateur.

```
        {
            /* cette fonction evalue si la transition peut etre  */
            /* mise en correspondance avec la sequence pointee   */
            /* le champ sequence du BLOC bloc_courant, et le cas */
            /* echeant genere le bloc suivant dans le chemin     */

            si correspondance (transition, bloc_courant, nouveau_bloc)
            {
                /* si le nouveau bloc constitue le dernier */
                /* maillon du chemin que l'on cherchait,   */
                /* alors la fonction renvoie VRAI          */
                si chemin_trouve (nouveau_bloc)
                    return VRAI;


                /* si le nouveau bloc est un bloc deja       */
                /* rencontre, que l'on est donc en train de  */
                /* cycler, mais que l'on evolue pas dans la  */
                /* recherche de la sequence specifiee par    */
                /* l'utilisateur, et si on ne constate aucune */
                /* superposition entre deux chemins, alors,  */
                /* on selectionne le nouveau bloc            */
                si((!cycle_inutile(nouveau_bloc) et (!element_table(nouveau_bloc,table_blocs)))
                    ajouter_en_debut_liste (liste_blocs_nouveaux, nouveau_blocs)

            }
        }

        /* on enleve le bloc que l'on vient d'utiliser de la liste */
        /* des blocs                                               */
        enlever_liste (liste_blocs_courants, bloc_courant);

        /* si la liste des blocs a explorer au niveau courant dans */
        /* l'automate est vide                                     */
        si (liste_blocs_courants == liste_vide)
        {
            /* si la liste des nouveaux blocs est vide, on n'a */
            /* donc pas trouve de chemin                       */
            si (liste_blocs_nouveaux == liste_vide)
                return FAUX;

            /* sinon on affecte a la liste des blocs courantes la */
            /* valeur de la liste des nouveaux blocs, et          */
            /* cette derniere repasse a NULL                      */
            sinon
                echanger_listes (liste_blocs_courant, liste_blocs_nouveaux);
        }
    }
    return FAUX;
}
```

- un champ prec de type BLOC, désignant le bloc précédent constituant le chemin en cours d'exploration
- un champ séquence correspondant à un pointeur pointant sur le groupe de labels courant dans la séquence fournie par l'utilisateur
- un champ fréquence de type ENTIER, désignant le nombre de fois où l'on a appliqué la règle courante dans la séquence spécifiée par l'utilisateur

L'algorithme peut alors s'écrire de la façon suivante

```
/* sequence : contient la sequence recherchee par l'utilisateur */

fonction plus_court_chemin (SEQUENCE sequence) : BOOLEEN

{
    /* liste_blocs_courants : liste des blocs qu'il reste a explorer au */
    /*                        niveau courant dans l'automate           */
    LISTE liste_blocs = liste_vide;

    /* liste_blocs_nouveaux : liste des blocs a explorer au niveau */
    /*                        suivant dans l'automate              */
    LISTE liste_blocs_nouveaux = liste_vide;

    /* bloc_courant : bloc que l'on est en train de visiter */
    BLOC bloc_courant;

    /* nouveau_bloc: bloc que l'on veut ajouter dans la liste des      */
    /*               nouveaux blocs a explorer                         */
    BLOC nouveau_bloc;

    /* transition : transition courante (label courant) */
    LABEL transition;

    /* table_blocs : table de hachage contenant l'ensemble des  BLOCS */
    /*               generes lors de la construction des chemins       */
    TABLE table_blocs;


    /* initialisation du bloc courant */
    bloc_courant = BLOC_INITIAL ();
    ajouter_dans_liste (liste_blocs_courants, bloc_courant);

    /* tant qu'il existe des blocs a explorer */
    tant que (liste_blocs_courants != liste_vide)
    {

        bloc_courant = premier_element_liste (liste_blocs);

        /* cette bouce permet de prendre en compte tous les labels */
        /* des arcs issus de bloc_courant.etat                     */
        pour transition dans transitions_suivantes(bloc_courant.etat) faire
```

Avec le nouveau format .seq, il peut exprimer le fait qu'entre deux **open** il ne veut pas de **close.**

Le format .seq simple est le format en sortie du nouvel Exhibitor (il est d'ailleurs important de remarquer qu'un fichier au format .seq simple est aussi au format .seq étendu), et il correspond donc à une description déterministe d'une séquence. Il sera pris en entrée par les programmes qui assurent la suite du passage de Caesar à Eludo.

V. La recherche de plus court chemin dans un automate

Exhibitor, dans OPEN/CAESAR, et Goal, dans Eludo, sont des outils qui permettent de rechercher une séquence de labels dans un LTS ou dans une spécification Lotos. L'intérêt de tels outils est essentiellement de pouvoir détecter la présence d'une séquence "interdite", qui permet ainsi de mettre en évidence une erreur contenue dans la spécification Lotos.

Jusqu'alors, Exhibitor présentait l'avantage de rechercher extrêmement rapidement des séquences de labels, mais ses deux défauts majeurs étaient, d'une part son implémentation récursive, qui présentait le risque d'une explosion de la pile durant l'analyse d'une spécification trop volumineuse, et d'autre part, la non-optimalité des résultats qu'il renvoyait. En effet, dans certains cas, Exhibitor pouvait détecter des séquences de 253 labels, correspondant à la description donnée par l'utilisateur, alors que certaines séquences, comprenant 20 labels, convenaient également.

Un des buts du projet était de modifier l'implémentation d'Exhibitor, en la rendant itérative, et de faire en sorte que les résultats renvoyés par cette nouvelle version soient de longueur optimale.

Par conséquent, nous allons voir, dans cette partie, quels choix ont été envisagés, pour atteindre ces objectifs et quelles solutions ont été adoptées.

V.1. L' algorithme de recherche de plus court chemin

Le parcours en largeur

Cet algorithme consiste à parcourir l'automate selon des couches d'états équidistantes de l'état initial (en considérant que la distance d'un sommet à la racine correspond à la longueur du chemin entre ces deux états, c'est-à-dire le nombre d'arcs du chemin permettant d'aller de ce sommet à la racine).

Considérons une structure de données appelée BLOC qui contienne:

- un champ état de type ETAT, désignant un sommet dans l'automate
- un champ label de type STRING, désignant le nom du label permettant d'accéder dans l'état état

IV.2.2. Le format .seq simple

Nous avons aussi décidé de conserver un format .seq, dit simple, dont l'utilité sera explicitée plus tard (cf partie 5).

Ce format .seq simple est identique, dans sa forme, au format .seq étendu, mais sa grammaire descriptive est allégée:

axiome	→	liste_de_sequences
liste_de_sequences	→	sequence
	|	sequence [] <retour_chariot> liste_de_sequences
sequence	→	liste_de_labels
liste_de_labels	→	groupe_de_labels <retour_chariot>
	|	groupe_de_labels <retour_chariot> liste_de_labels
groupe_de_labels	→	label
label	→	label_elementaire
label_elementaire	→	chaine_simple
	|	"chaine_C"

De façon informelle, ceci revient à dire, que le format .seq simple est le format .seq original dans lequel on a ajouté des guillemets en début et en fin de chaque label.

IV.3. Utilisation du nouveau format .seq.

Le format .seq étendu est celui utilisé en entrée du nouvel Exhibitor. Il donne une plus grande liberté à l'utilisateur, dans la description des séquences qu'il recherche.

Ainsi, avec ce langage, il peut exprimer le non-respect de l'accés en exclusion mutuelle à une ressource, ce que ne permettait pas l'ancien format. Il le fera de la manière suivante :

```
~"OPEN !STATION1" *
"OPEN !STATION1"
~"OPEN !STATION2" & ~"CLOSE STATION1" *
"OPEN !STATION2"
~"CLOSE !STATION1" & ~"CLOSE STATION2" *
"CLOSE !STATION1"
~"CLOSE !STATION2" *
"CLOSE !STATION2"
```

expression_reguliere_Unix_non_vide : chaîne de caractères non vide correspondant à la description faite dans le manuel de la commande ed (cf man ed).

~ : opérateur de négation vis-à-vis d'un label élémentaire. La sémantique de cet opérateur est d'indiquer que l'on accepte tout label_elementaire sauf celui précédé par l'opérateur ~.

& : opérateur et entre un label_elementaire et un label. La sémantique de cet opérateur est, qu'un label donné peut être mis en correspondance (to match, en anglais) avec le label, s'il peut être mis en correspondance avec chacun des deux termes du &.

| : opérateur ou entre un label_elementaire et un label. La sémantique de cet opérateur est, qu'un label donné peut être mis en correspondance avec le label, s'il peut être mis en correspondance avec au moins un des deux termes du |.

^ : opérateur ou exclusif entre un label_elementaire et un label. La sémantique de cet opérateur est, qu'un label donné peut être mis en correspondance avec le label, s'il peut être mis en correspondance avec un et seulement un des deux termes du ^.

* : opérateur de répétition d'un groupe_de_labels. La sémantique de cet opérateur sera vue par la suite.

+ : opérateur de répétition (au moins une fois) d'un groupe_de_labels. La sémantique de cet opérateur sera vue par la suite.

<deadlock> : indicateur de deadlock (état puits). La sémantique de ce symbole est que l'état dans lequel on se trouve ne doit pas posséder de transitions sortantes. Cet identificateur, permet donc d'apporter une nouvelle fonctionnalité à Exhibitor, qui est de rechercher une séquence se terminant pas un état puits.

[] : séparateur de séquences

<retour_chariot> : caractère de retour chariot

Exemple: dans ce format .seq étendu, l'exemple donné pour le format .seq original, peut se ré-écrire de la façon suivante

```
[MONEY*]
COFFEE | TEA
[]
WATER
```

Dans ce format .seq étendu, les commentaires sont toujours contenus entre CTRL-A et CTRL-B.

- pour permettre à l'utilisateur de ne pas avoir à connaître la valeur de toutes les offres associées à une porte, ainsi que leur nombre, nous avons décidé d'introduire les expressions régulières Unix dans le format .seq.

Cependant, et afin de rester dans le prolongement du projet EUCALYPTUS, nous avons tout fait pour que le format .seq original soit compatible avec le nouveau. Ainsi, les modifications à apporter aux fichiers à l'ancien format .seq sont minimes: il suffit de rajouter le caractère **"** en début et en fin de chaque label.

IV.2.1. Le format .seq étendu

Nous avons alors élaboré la grammaire de description de ce langage de la façon suivante:

axiome	→	liste_de_sequences
liste_de_sequences	→	sequence
	|	sequence [] <retour_chariot> liste_de_sequences
sequence	→	liste_de_labels
liste_de_labels	→	groupe_de_labels <retour_chariot>
	|	groupe_de_labels <retour_chariot> liste_de_labels
	|	<deadlock> <retour_chariot>
groupe_de_labels	→	label
	|	label *
	|	label +
label	→	label_elementaire & label
	|	label_elementaire | label
	|	label_elementaire ^ label
	|	label_elementaire
label_elementaire	→	chaine_simple
	|	"chaine_C"
	|	[expression_reguliere_Unix_non_vide]
	|	(label)
	|	~ label_elementaire

Description des terminaux de la grammaire:

chaine_simple : chaîne de caractères ne comprenant que des chiffres, des lettres (majuscules ou minuscules) ainsi que le caractère '_' (tiret bas).

chaine_C : chaîne de caractères au sens de la grammaire du langage C.

commençant et finissant par un même label et de longueur différente de deux.

D'autre part, il n'était pas possible "d'ignorer" la valeur des offres associées a une porte, ou de ne pas en connaître le nombre.

Pour mieux voir les limites de ce langage, prenons un exemple concret. Un utilisateur cherche dans une spécification donnée, un non-respect de l'accés en exclusion mutuelle à une ressource, dans un cas où il y a deux clients. Il cherche donc la séquence suivante :

```
OPEN !STATION1
OPEN !STATION2
CLOSE !STATION1
CLOSE !STATION2
```

Dans cette séquence il y a deux **open** de suite, ce qui traduit le non-respect des conditions d'accés (exclusion mutuelle) à la ressource.

Nous ne pouvons pas préciser que entre les deux **open** nous ne voulons pas un **close** . Ce qui fait que même avec une spécification correcte, l'ancien Exhibitor trouvera un chemin correspondant à la séquence fournie. Il fournira un chemin comme celui-ci :

```
OPEN !STATION1
CLOSE !STATION1
OPEN !STATION2
CLOSE !STATION2
OPEN !STATION1
CLOSE !STATION1
OPEN !STATION2
CLOSE !STATION2
```

L'utilisateur ne parvient donc pas avec le format .seq actuel à exprimer le non respect de l'exclusion mutelle.

Il y parviendra, avec le nouveau format .seq, puisque nous fournissons un langage qui permet de mieux décrire la séquence cherchée, ce que nous allons maintenant étudier.

IV.2. Les modifications apportées au format .seq

Un premier but de ce projet était donc d'étendre les possibilités offertes par le format .seq.

Ainsi, nous avons cherché à palier aux trois lacunes précédemment évoquées, de la manière suivante:

- afin de permettre une spécification partielle (ou indéterministe) de la séquence recherchée, nous avons introduit:
  - deux opérateurs de répétition notés * (répétition éventuelle) et + (répétition avec au moins une occurrence)
  - trois opérateurs binaires sur les labels, notés & (et), | (ou) et ^ (ou exclusif)
  - un opérateur unaire, noté ~ (négation).

est un peu différent sous ELUDO.

Parmi ces outils, Exhibitor est un programme permettant d'effectuer la recherche d'un chemin (ou séquence) dans un LTS. Il produit, en sortie, un fichier au format .seq qui décrit le chemin qui a été parcouru dans l'automate, et qui correspond à la séquence recherchée.

Pour ce faire, Exhibitor prend en entrée un LTS (ou une spécification Lotos) ainsi que la description de la séquence fournie par l'utilisateur, dans un format de données appelé format .seq.

Dans cette partie nous allons donc voir quelles ont été les modifications et les améliorations que nous avons apportées au format .seq.

IV.1. Le format .seq original

IV.1.1. Description

A l'origine, un fichier au format .seq comprenait une ou plusieurs séquences.

Deux séquences étaient séparées entre elles, par un symbole noté [] qui devait se trouver seul sur une ligne du fichier.

Une séquence comprenait un ou plusieurs labels, séparés par un caractère de retour chariot.

Chaque label correspondait à une chaîne de caractères correspondant à une porte Lotos, munie éventuellement de ses offres.

Exemple de fichier au format .seq :

```
MONEY !2
TEA
[]
MONEY !3
COFFEE
[]
WATER
```

Ce fichier comprend trois séquences comprenant respectivement deux, deux et un label.

Il était possible d'insérer des commentaires dans un fichier au format .seq : un commentaire commençait par un CTRL-A (code 001 en ASCII) et se terminait par un CTRL-B (code 002 en ASCII).

IV.1.2. Les limites d'un tel format

Les limites de ce premier format .seq étaient importantes. En effet, il n'était absolument pas possible de rechercher un chemin dont on ne connaissait pas tous les labels, par exemple

plus complexe (le système téléphonique décrit dans le fichier POTS.l), l'analyse de trace sous TETRA fonctionnait correctement sous une version (version 0.9992) antérieure à la version courante de l'IHM d'EUCALYPTUS, mais la ré-exécution du fichier checkpoint (.cp) obtenu en sortie de TETRA n'a pas fonctionné. Nous n'avons jamais réussi à faire fonctionner TETRA sous des versions ultérieures à la version 0.9992 de l'IHM d'EUCALYPTUS.

III.3. La ré-exécution de traces

Qu'entendons nous par ré-exécution ?

Lorsqu'on simule une spécification Lotos à l'aide d'Isla, il apparaît une fenêtre HISTORY qui montre le chemin d'exécution parcouru lors de la simulation. Il est possible, en cliquant sur un des noeuds de ce chemin, de revenir dans l'état correspondant et de faire un autre choix ou de ré-exécuter tel quel le chemin déjà parcouru. Un exemple de fenêtre HISTORY constitue l'annexe IX.2 .

On peut sauvegarder les résultats de la simulation (dans un fichier checkpoint) sous Isla. Lorsque l'on recharge cette séance de simulation, on se retrouve dans la configuration précédant la sauvegarde.

Pour pouvoir ré-exécuter une trace de simulation (provenant d'Isla ou de CAESAR) sous Isla, il faudrait obtenir la fenêtre HISTORY dans laquelle serait le chemin d'exécution parcouru lors de la simulation précédente.

III.4. Extension de l'interface

Pourquoi se limiter à une trace ?

Il nous a semblé intéressant d'intégrer à notre interface, un outil de recherche de séquences dans une spécification Lotos. Ainsi, avec notre interface, l'utilisateur pourra utiliser la rapidité de CAESAR pour trouver une sequence dans sa spécification, puis utiliser ELUDO pour comprendre comment celle-ci a été obtenue.

Il existe déjà un outil de recherche de séquences qui s'appelle *Exhibitor*, et qui appartient à OPEN/CAESAR. Nous nous sommes proposés de l'améliorer, notamment en changeant le langage d'entrée de celui-ci, pour apporter une plus grande liberté à l'utilisateur, (cf le chapître suivant), et en modifiant l'algorithme de recherche de séquences de manière à ce qu'il renvoie des solutions optimales (cf chapître 5).

IV. L'extension du format .seq

Une spécification Lotos peut être vue comme un LTS (labelled transitions system) autrement appelé automate. Les outils contenus dans OPEN/CAESAR permettent d'effectuer un certain nombre d'opérations sur des LTS et, par conséquent, sur des spécifications Lotos. Ceci

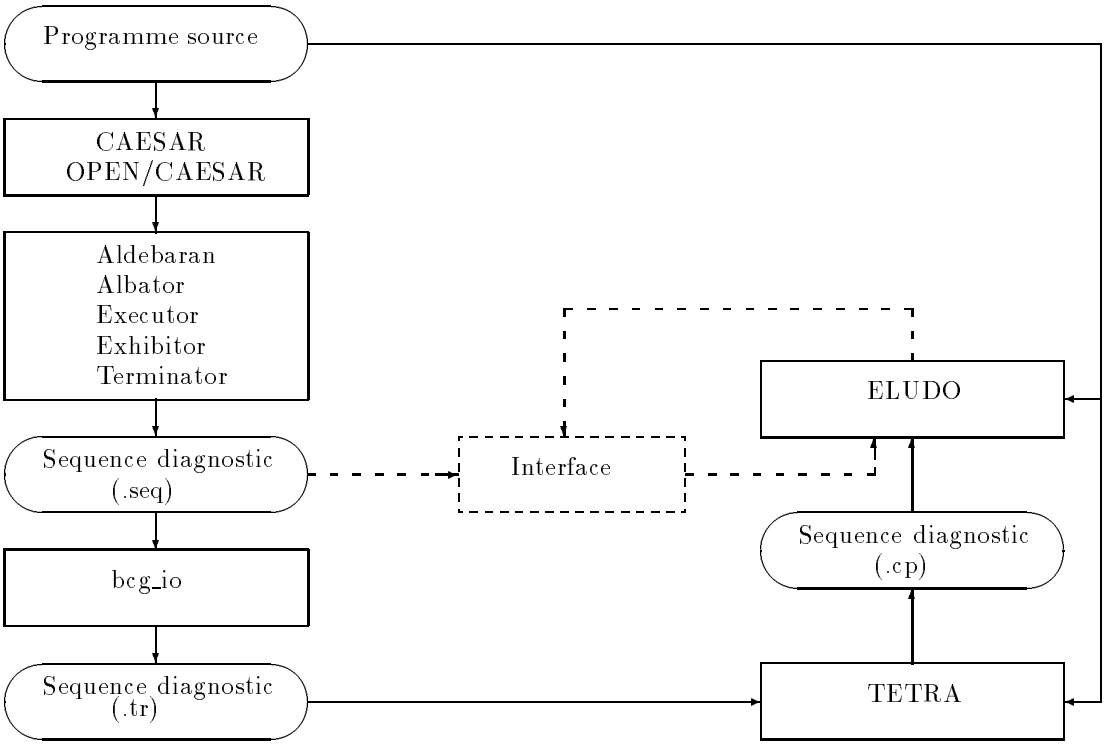
d'ELUDO. Ainsi, l'utilisatuer tire profirt des avantages de ces outils.

Il pourrait utiliser CAESAR, pour trouver rapidement une anomalie dans sa spécification (un deadlock, par exemple), rejouer les traces laissées par cette recherche avec ELUDO, et comprendre exactement pourquoi il y a une anomalie dans sa spécification.

Le problème est que les fichiers de traces produits par OPEN/CAESAR, et plus précisément par l'outil Exhibitor, ne sont pas écrits dans le même format que ceux de ELUDO, et ils ne contiennent pas les mêmes informations, d'où le besoin d'une interface entre les deux:

Fichier au format .seq		Fichier en entrée
généré par OPEN/CAESAR ---------->	Interface -------->	de ELUDO

La figure ci-dessous décrit l'ensemble des communications qui existaient ou que nous avons ajoutées (flèche en pointillés) entre CAESAR et ELUDO.



Communication CAESAR-ELUDO

Nous notons sur ce schéma, la possibilité de convertir les fichiers de traces (suffixe .seq) en des fichiers checkpoint (suffixe .cp) lisibles par ELUDO, en passant le programme TETRA. Nous avons essayé cette passerelle. Pour un programme Lotos simple (le distributeur de boissons), nous avons réussi à ré-exécuter une trace par ELUDO. Pour une spécification Lotos

- TETRA développé à Montréal
- ELUDO, ISLA, SELA et GOAL développés à Ottawa

Parmi ces ensembles d'outils constituant le projet EUCALYPTUS, nous nous intéresserons plus particulièrement à ceux développés à Grenoble et à Ottawa. Plus précisément, dans ELUDO, notre attention se portera sur les deux modules que sont ISLA et GOAL, et dans CAESAR, sur Exhibitor:

- *ISLA*: permet une exécution interactive pas à pas d'une spécification Lotos
- *GOAL*: trouve le chemin d'exécution aboutissant à un but donné, le but étant une occurrence d'une porte Lotos
- *Exhibitor*: recherche une séquence de labels (représentant des transitions), dans une machine d'états finis

Notre projet consistait donc en une unification des outils existants :

- une même interface homme-machine (IHM) intégrant l'ensemble des outils
- une unification des outils CAESAR et ELUDO

III.2. L'intérêt de l'unification de CAESAR et ELUDO

Ces outils de vérification de spécifications Lotos (CAESAR et ELUDO) traduisent une spécification Lotos avant de pouvoir la traiter. Cependant, il existe une nette différence entre les types de traduction préférés par CAESAR et ELUDO: le premier optant pour une machine d'états finie et le second pour une base de données Prolog. Il en découle également des caractéristiques d'utilisation très différentes:

- CAESAR permettant un traitement rapide mais ne pouvant pas faire le rapprochement entre les résultats et la spécification.

- ELUDO offrant un traitement plus lent, mais disposant de l'information pertinente pour établir le lien entre les résultats et la spécification originale.

D'autre part, CAESAR effectue tout le traitement par lots, donc avec peu d'interactions avec l'utilisateur, alors que les programmes d'ELUDO sont pour la plupart interactifs. Avec ISLA, par exemple, l'utilisateur doit préciser à chaque étape de la simulation, quelle est la prochaîne action à effectuer.

Signalons au passage que CAESAR a été intégralement codé en C, alors que ELUDO est entièrement écrit en Prolog.

ELUDO étant particulièrement habilité à l'analyse interactive et CAESAR au traitement par lot, il semble à propos d'ajouter à ELUDO la capacité de rejouer une trace générée par CAESAR. Cette fonctionnalité est nécessaire et suffisante pour permettre au spécificateur d'utiliser conjointement les spécificités de chaque outil: rapidité de CAESAR et précision

II.3.2. Rôle du TSERG

L'activité principale de ce groupe consiste en la spécification, l'élaboration de tests et la vérification de protocoles et de logiciels de communication. Cependant, les thèmes de recherche suivants sont également abordés:

- l'application de systèmes experts à la conception de logiciels de communication
- l'étude des problèmes liés à l'élaboration et à la phase de tests de logiciels de gestion des réseaux téléphoniques

Le but du TSERG étant d'être parmi les laboratoire en pointe dans le domaine des télécommunications, il a développé de nombreux projets internationaux, pour l'aide à la spécification de protocoles, notamment.

Enfin, signalons que les laboratoires canadiens étant obligés de s'auto-financer, le TSERG a de nombreux contacts avec l'industrie afin d'orienter, dans la mesure du possible, sa recherche en fonction des besoins de celle-ci.

III. Le projet EUCALYPTUS

III.1. Présentation générale

Le développement des protocoles de télécommunication et des systèmes distribués peut être grandement amélioré et simplifié par l'utilisation de méthodes formelles et d'outils logiciels les supportant. L'objectif du projet EUCALYPTUS était de confectionner une boite à outils pour l'ingénierie des protocoles, dédiée au langage Lotos.

Ce projet euro-canadien est le fruit d'une collaboration entre plusieurs laboratoires de recherche:

- l'Inria Rhône-Alpes (VERIMAG) à Grenoble (France)
- l'Université de Liège (Belgique)
- le Département Informatique de l'Université d'Ottawa (Canada)
- l'Université de Montréal (Canada)

EUCALYPTUS est donc un ensemble d'outils, qui permettent de simuler et de vérifier des spécifications rédigées dans le langage Lotos (norme ISO, établie pour la spécification des protocoles). Il consiste en plusieurs ensembles logiciels, qui ont été développés ou améliorés lors du projet:

- ALDEBARAN, BCG, CAESAR, CAESAR.ADT et OPEN/CAESAR développés à Grenoble
- APERO développé à Liège

II.2. Le Département d'Informatique de l'Université d'Ottawa

Le Département d'Informatique de l'Université d'Ottawa a été créé en 1970. Il délivre, par le biais de l'Université, les diplômes suivants: BSc, MCS et PhD. Ses chercheurs travaillent aussi bien dans le domaine de la recherche appliquée que de la recherche fondamentale.

Les secteurs de recherches abordés dans ce Département sont:

- les algorithmes combinatoires appliqués
- la théorie des graphes
- le graphisme
- l'élaboration de protocoles de communication
- les systèmes d'information
- la théorie de l'informatique distribuée
- les tests de logiciels et de protocoles
- la reconnaissance de la langue naturelle
- ...

Les principales institutions qui financent la recherche du Département sont NSERC, Bell Canada, Bell-Northern Research, Cognos, Gandalf, le Département de Communications, l'Institut de Recherche en Télécommunications de l'Ontario et l'Institut Canadien pour la Recherche en Télécommunications.

En outre, le Département édite également des publications scientifiques et organise des conférences internationales.

Durant les dernières années, le nombre de thésards issus du Département a sensiblement augmenté, ce qui a entraîné une croissance de son activité.

II.3. Le Groupe de Recherche en Ingénierie de Logiciels de Télécommunications

II.3.1. Composition du groupe

Le TSERG (Telecommunications Software Engineering Research Group) comprend sept professeurs (T.Y. Cheung, L. Logrippo, A. Mili, R. L. Probert (coordinator), J. Raymond, H. Ural, G. White) du Département d'Informatique de l'Université d'Ottawa. D'autre part, les professeurs R. Buhr, M. Woodside et G. Karam du Département de Systèmes et d'Informatique de l'Université de Carleton ainsi que des personnes issues de l'industrie font partis du TSERG.

Six Ph.D. et une vingtaine de Master sont encadrés dans leur travail, au sein du groupe.

II. Le cadre du projet

II.1. L'Université d'Ottawa

II.1.1. Historique

Fondée en 1848, l'Université d'Ottawa était, à l'origine, un "Roman Catholic College". En 1965, elle entre dans le groupe des campus publics de l'Ontario. De nos jours, l'Université d'Ottawa ne cesse de remplir ses deux missions de base, que sont le développement de la recherche et l'enseignement supérieur.

En tant que première université bilingue du Canada, elle offre à tous ses étudiants, qu'ils soient canadiens ou étrangers, un mélange culturel unique.

II.1.2. Missions de l'Université d'Ottawa

Située dans la capitale canadienne, l'Université d'Ottawa se trouve, géographiquement et culturellement, à la frontière des parties francophones et anglophones du Canada. D'autre part, sa proximité des agences gouvernementales et des centres de recherche lui confère un rôle de lien entre le milieu de l'enseignement et le monde du travail.

Afin de préserver et d'étendre sa renommée, l'Université d'Ottawa se doit d'assurer la meilleure qualité possible, aussi bien au niveau des enseignements qu'elle dispense, que des recherches qui y sont menées dans des domaines aussi variés que l'administration, les arts, l'enseignement, la médecine et les domaines médicaux et para-médicaux, le droit, les sciences et les sciences sociales.

Les buts de l'Université d'Ottawa sont :

- de toujours élargir et approfondir le champ de ses enseignements et de ses recherches
- d'assurer la pérennité du bilinguisme en son sein
- d'occuper une place prédominante dans le développement de nouvelles méthodes d'enseignement et de recherche, notamment vis-à-vis des étudiants francophones d'Ontario
- de demeurer l'une des universités les plus en pointe, dans le domaine de la présence de la gente féminine dans tous les aspects de la vie sociale universitaire
- de s'ouvrir toujours plus aux relations internationales.

I. Introduction

Ce document constitue le rapport de projet de fin d'études à l'ENSIMAG, que nous avons effectué du 29 mars au 19 juin 1996, dans le Département d'Informatique de l'Université d'Ottawa (Canada), dirigé par le Professeur Luigi Logrippo. Nos responsables, durant ce stage, étaient Monsieur Jacques Sincennes, pour ce qui est de la partie canadienne, et Monsieur Hubert Garavel, pour ce qui est de l'ENSIMAG et de l'Inria Rhône-Alpes (VERIMAG).

Le cadre de ce stage était le projet EUCALYPTUS. Ce dernier a été développé communément par le Département d'Informatique de l'Université d'Ottawa et l'Inria Rhône-Alpes. Son but est d'offrir un ensemble d'outils permettant de mettre au point et de corriger des spécifications de protocoles temps réel, écrites dans le langage de spécification Lotos. Le projet EUCALYPTUS a donc abouti à la production de différents outils, tels les programmes d'OPEN/CAESAR (ensemble logiciel développé par l'Inria Rhône-Alpes) et d'ELUDO (ensemble logiciel développé par le Département Informatique de l'Université d'Ottawa) entre autres, ainsi qu'à leur intégration dans une interface graphique commune.

Les buts de notre projet étaient:

- de rendre plus performants certains outils d'OPEN/CAESAR et d'ELUDO, aussi bien d'un point de vue temps d'exécution que d'un point de vue puissance de résolution.

- d'améliorer et/ou d'élargir la communication entre l'ensemble de programmes fourni par OPEN/CAESAR et celui proposé par ELUDO.

- d'écrire une interface graphique plus ergonomique, plus rapide et plus agréable pour l'utilisateur, pour la boite à outils EUCALYPTUS.

- de ré-écrire, dans un langage plus adapté, l'interface graphique d'un outil d'OPEN/CAESAR appelé X-Simulator.

Par conséquent, après une présentation rapide de l'Université d'Ottawa et de son Département d'Informatique, nous ferons une description détaillée du projet EUCALYPTUS, puis nous aborderons les améliorations que nous avons apportées au format de données .seq, utilisé par les outils d'OPEN/CAESAR. Dans la partie suivante, nous présenterons les modifications faites au niveau des programmes de recherche de chemin dans un automate (Exhibitor et Isla). Ensuite, nous parlerons des nouvelles possibilités de communication entre OPEN/CAESAR et ELUDO. Avant de conclure ce rapport, nous décrirons, dans une dernière partie, les nouveaux outils graphiques.

VII.1.3.Les limites 34
VII.2.La nouvelle IHM 34
VII.2.1.L'outil de développement Tcl/Tk 35
VII.2.2.Aspect extérieur 36
VII.2.3.Caractéristiques internes de l'interface 37
VIII.Conclusion 39
IX.Annexes 40
IX.1.Tests d'exhibitor 40
IX.1.1.Test 1: 40
IX.1.2.Test 2 41
IX.1.3.Test 3 41
IX.2.Exemple d'utilisation de la passerelle : 46
IX.3.L'ancienne Interface Homme-Machine 47
IX.4.La nouvelle Interface Homme-Machine 48

Table des Matières

I.Introduction 5
II.Le cadre du projet 6
II.1.L'Université d'Ottawa 6
II.1.1.Historique 6
II.1.2.Missions de l'Université d'Ottawa 6
II.2.Le Département d'Informatique de l'Université d'Ottawa 7
II.3.Le Groupe de Recherche en Ingénierie de Logiciels de Télécommunications 7
II.3.1.Composition du groupe 7
II.3.2.Rôle du TSERG 8
III.Le projet EUCALYPTUS 8
III.1.Présentation générale 8
III.2.L'intérêt de l'unification de CAESAR et ELUDO 9
III.3.La ré-exécution de traces 11
III.4.Extension de l'interface 11
IV.L'extension du format .seq 11
IV.1.Le format .seq original 12
IV.1.1.Description 12
IV.1.2.Les limites d'un tel format 12
IV.2.Les modifications apportées au format .seq 13
IV.2.1.Le format .seq étendu 14
IV.2.2.Le format .seq simple 16
IV.3.Utilisation du nouveau format .seq. 16
V.La recherche de plus court chemin dans un automate 17
V.1.L' algorithme de recherche de plus court chemin 17
V.2.Choix de l'algorithme et implémentation: Le nouvel Exhibitor 20
V.3.La fonction de correspondance 20
V.4.Exemple d'utilisation du nouveau Exhibitor 23
VI.La communication entre CAESAR et ELUDO 24
VI.1.Intégration des résultats de Exhibitor dans ELUDO 25
VI.1.1.Architecture de cette passerelle 26
VI.1.2.Production de fichiers "checkpoint" 26
VI.1.3.Les DDP 26
VI.1.4.Production des DDP 27
VI.1.5.Validité du chemin 28
VI.1.6.Les Expériences 29
VI.1.7.Architecture de l'implémentation 29
VI.1.8.Tests 30
VI.1.9.Exemple d'utilisation de cette passerelle 31
VI.2.Deuxième solution: Production de "Checkpoint" à partir de ".seq étendu" 31
VI.3.Troisième solution: Combinaison de Exhibitor et GOAL 33
VII.L'interface d'EUCALYPTUS 33
VII.1.L'IHM existante 33
VII.1.1.Développement 33
VII.1.2.Aspects extérieurs 34

Remerciements

Nous tenons à remercier toutes les personnes qui nous ont permis de mener à bien ce projet, et de faire en sorte qu'il se déroule dans les meilleures conditions possibles.

Plus particulièrement, nous tenons à remercier :

Monsieur le Professeur Luigi LOGRIPPO, responsable du Département d'Informatique de l'Universite d'Ottawa

Monsieur Hubert GARAVEL, enseignant à l'E.N.S.I.M.A.G. et chercheur à l'Inria Rhône-Alpes (Verimag)

Monsieur Jacques SINCENNES, chercheur et ingénieur système du Département d'Informatique de l'Université d'Ottawa.

D'autre part, nous remercions toutes les personnes du Département d'Informatique de l'Université d'Ottawa ainsi que celles de l'Inria qui nous ont aidés dans notre travail.

Xavier ETCHEVERS
Jean-Michel FRUME
Mark JORGENSEN

Xavier ETCHEVERS
Jean-Michel FRUME
Mark JORGENSEN

20 Juin 1996

DE CAESAR A ELUDO
Projet de troisième année à l'E.N.S.I.M.A.G. dans le cadre du projet EUCALYPTUS

Responsables : **Luigi LOGRIPPO**
Hubert GARAVEL
Jacques SINCENNES