CPPM Centre de Physique des Particules de Marseille

Thèse de Doctorat présentée par

QIAN Zuxuan

pour obtenir le grade de Docteur de l'Université Aix-Marseille II,
Faculté des Sciences de Luminy,
spécialité Physique théorique et des particules élémentaires

Architecture distribuée pour un logiciel d'analyse interactive en physique des particules

Soutenue le 21 Septembre 1990 devant la commission d'examen :

Messieurs :

J.V. Allaby
J.J. Aubert, Président
A. Bonissent
F. Etienne
P. Palazzi
M. Van Caneghem

CPPM Centre de Physique des Particules de Marseille

Thèse de Doctorat présentée par

QIAN Zuxuan

pour obtenir le grade de Docteur de l'Université Aix-Marseille II,
Faculté des Sciences de Luminy,
spécialité Physique théorique et des particules élémentaires

Architecture distribuée pour un logiciel d'analyse interactive en physique des particules

Soutenue le 21 Septembre 1990 devant la commission d'examen :

Messieurs :

J.V. Allaby
J.J. Aubert, Président
A. Bonissent
F. Etienne
P. Palazzi
M. Van Caneghem

Avant-propos

Ce travail s'inscrit dans le cadre de la collaboration ALEPH, du nom de l'un des quatres détecteurs de l'accélérateur LEP au CERN (Genève).

L'analyse de données de physique des particules est une procédure de réduction complexe de données : des millions d'événements sont condensés progressivement jusqu'à quelques chiffres qui confirment ou infirment certaines prédictions théoriques. Pour effectuer ce travail, de nombreux logiciels d'analyse de données sont développés soit pour une utilisation générale (ex. manipulation d'histogrammes) soit pour l'analyse spécifique de données des expériences. Ils sont habituellement écrits en FORTRAN et organisés en bibliothèque de procédures appelées par l'utilisateur. Le physicien, pour effectuer son analyse, a besoin de connaissances sur les structures de données traitées et sur les composants des bibliothèques de procédures.

Le projet PIGAL (Prolog Interactive and Graphic for ALEPH analysis) vise à construire un environnement interactif permettant d'assister le physicien dans son travail : visualiser et analyser événement par événement, ou bien effectuer l'analyse statistique, en le libérant de la connaissance détaillée de la structure de données, ainsi que des aspects fastidieux de l'écriture des programmes. Dans cet environnement le physicien trouve les outils pour l'analyse de données, par ex. sélection des événements, représentation graphique 2D/3D, histogrammation de variables physiques, et aussi pour la mise au point de logiciels d'analyse, par ex. test d'algorithmes, surveillance de changement de programmes.

Certains problèmes sont apparus pendant le développement, notamment celui de l'intégration de logiciels développés dans d'autres contextes, comme le programme de représentation d'histogramme et le programme d'analyse physique. Le problème d'incompatibilité des ressources de ces

logiciels a été résolu au niveau de l'architecture du logiciel : l'architecture répartie. Cette expérience a montré que l'utilisation de techniques distribuées dans le domaine des systèmes d'analyse est efficace. Ce travail permet d'envisager une architecture plus génerale pour des logiciels hétérogènes complexes avec une structure extrêmement simple.

Un autre problème est lié à l'évolution très rapide des matériels graphiques, qui impose d'adapter souvent le programme, ce qui peut être plus facilement fait dans un environnement distribué.

Le logiciel PIGAL a été conçu à Marseille et réalisé par une équipe de 9 personnes avec une large contribution de différents laboratoires.

J'ai participé au développement du système, concernant : l'interface PIGAL avec le programme de reconstruction, le progiciel d'histogrammes, la sélections d'événement, l'intégration du logiciel de représentation d'histogramme, l'intégration du logiciel de représentation des conditions de déclenchement (trigger), et finalement la conception et sa réalisation de l'architecture distribuée.

Remerciements

Je tiens tout particuliérement à remercier le Professeur Jean-Jacques Aubert pour m'avoir accueilli dans son laboratoire et installée dans des conditions idéales de travail.

Messieurs Alain Bonissent et François Etienne m'ont dirigé tout au long de ce travail. Leurs suggestions sur tous les domaines abordés ont largement contribué à la réalisation de ce projet. Je les remercie pour leur encouragement, patience et disponibilité.

Je tiens à exprimer toute ma gratitude à Monsieur Paolo Palazzi pour des conseils sincères et de l'aide amical qu'il m'a donnés depuis toujours. Je le remercie, ainsi que Messieurs James Allaby, Jean-Jacques Aubert, Alain Bonissent, François Etienne et Michel Van Caneghem pour avoir accepté de participer au jury de cette thèse.

J'adresse tous mes remerciements à Messieurs Tim Berners-Lee et Alberto Aimar, sans qui le prototype du système réparti n'aurait pas vu le jour.

Je sais gré à Monsieur Patrice Payre d'avoir corrigé, avec une patience à toute épreuve, ce manuscrit écrit en ma troisième langue etrangère.

Mes remerciements vont aussi à tous les membres du laboratoire du Centre de Physique des particules de Marseille pour l'ambiance qu'ils créent et au sein duquel il est très agréable de travailler.

Table des matières

Liste des Figures

Chapitre 1

ALEPH, une expérience de physique des particules au LEP

Le LEP, situé au CERN (près de Genève), est le collisionneur électron-positron de la plus haute énergie au monde. Il a la forme d'un cercle d'environ 27 km de circonférence, et sa profondeur sous terre varie entre 45 et 175 m. La machine est en mesure d'accélérer les électrons et les positrons, dans des directions opposées, à des énergies pouvant atteindre 100 GeV par faisceau, ce qui correspond à une énergie de 200 GeV disponible par collision.

Les faisceaux d'électrons et positrons circulent séparément dans un même tube à vide (10^{-11} Torrs), et entrent en collision en quatre points du cercle, où se trouvent les grands détecteurs de particules (ALEPH, L3, DELPHI, OPAL) qui sont en mesure d'étudier la nature et la fréquence des différents processus mis en oeuvre dans de telles collisions. Le LEP est construit pour une luminosité de quelque 10^{31} cm^{-2} s^{-1} .

Les recherches expérimentales effectuées au LEP se situent dans le cadre théorique du modèle standard. Après la mise en évidence au CERN en 1983 des bosons W et Z^o, les expériences du LEP visent à étudier avec précision les propriétés de ces particules, afin de préciser les paramètres du modèle standard, et de vérifier avec une grande précision ses prédictions.

Dans la première version (LEP I), les objectifs sont liés à la physique du Z^o :

- **la masse et la largeur du Z^o .**
  La mesure précise permet en particulier de fixer le nombre de familles de leptons et de préciser les paramètres du modèle standard, par exemple $\sin^2\theta_W$ et les constantes de couplage vectorielles et axiales.

- **la découverte de particules standards nécessaires au modèle**: le quark Top, le Higgs.

- **la détection de particules nouvelles** : leptons lourds, particules supersymmétriques, particules de charges fractionnaires, etc.

- **l'étude de l'interaction forte** par étude des jets ou la mesure de la constante de couplage de cette interaction.

LEP II atteindra une énergie de 200 GeV, permettant probablement de mettre en évidence le quark Top et d'étudier la production de particules $W^{\pm}$, et leur couplage avec Z^o[1,2,3].

Le LEP a été mis en route en Août 1989, le nombre des événements enregistrés par ALEPH est présenté dans la table 1.

Table 1 : Statistique des événements enregistrés dans ALEPH

	1990	1989	Total
Triggers :	685347	468517	1153864
Z^o :	46863	29704	76567
Bhabhas:	73641	54284	127925

Les principaux résultats de l'analyse portant sur les données de 1989 sont:

- détermination précise de la masse et de la largeur du Z^o [4]:

  $Mz = 91.182 \pm 0.026(exp.) \pm 0.030(faisceau)$ GeV, $\Gamma z = 2.541 \pm 0.056$ GeV

  ce qui permet de déterminer la section efficace de desintégration en deux neutrinos, et donc de fixer le nombre de neutrinos [4]:

  $N = 3.01 \pm 0.15$

- Vérification de l'universalité des leptons [5]: les trois leptons (électron, muon, tau) ont le même couplage avec le Z^o. Leurs largeurs partielles sont identiques : 85.4 ± 5.3 Mev

- Détermination des taux de branchement [6]

  $Z^o \rightarrow b\bar{b}$ (0.215), $Z^o \rightarrow c\bar{c}$ (0.148)

- des limites ont été fixées sur la masse du boson de Higgs (exclus de l'intervalle **[32Mev-24Gev]**) [7]

1.1 Détecteur ALEPH

ALEPH est l'une de quatre expériences au LEP [8,9]. Le détecteur d'ALEPH, avec $12 \times 12 \times 12$ m^3 de dimensions et une masse de 3000 tonnes, comporte six différentes couches de détecteurs, représentées Fig.1,2,3. Ce sont, de l'intérieur vers l'extérieur :

- le détecteur de Vertex au silicium
- la Chambre à Traces Interne (ITC)
- la Chambre à Projection Temporelle (TPC)
- le Calorimètre Electromagnétique (ECAL)
- le Calorimètre Hadronique (HCAL)
- la Chambre à Muons

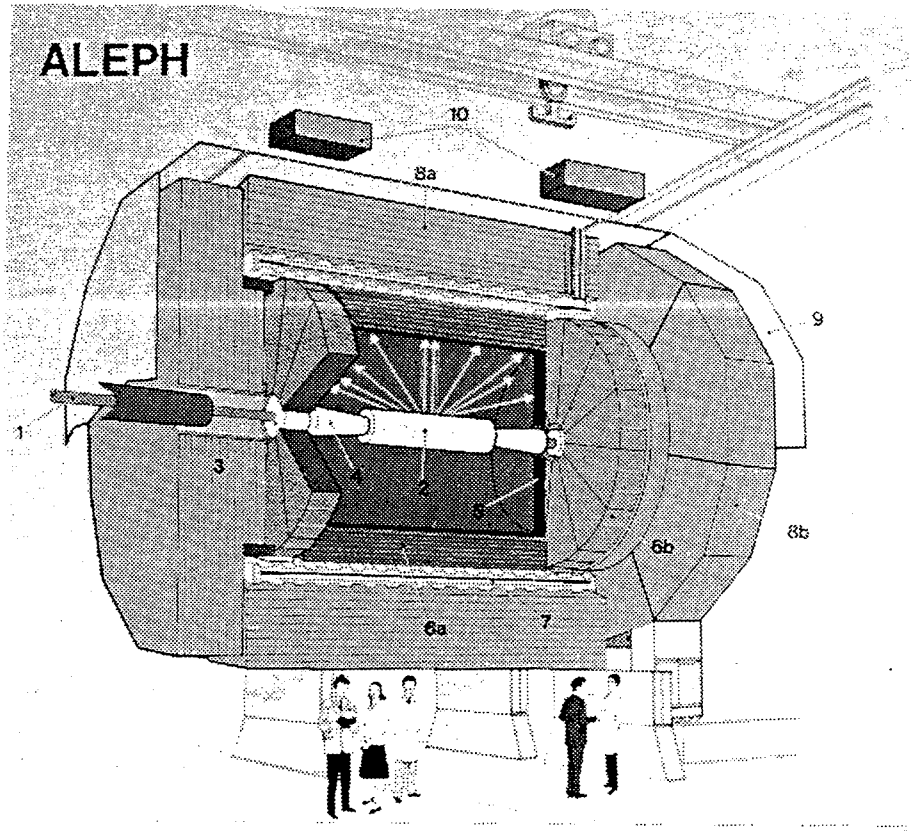


Fig. 1 Détecteur ALEPH
1. Ligne de faisceau
2. Chambre à Traces Interne (ITC)
3. Moniteur de luminosité
4. Cône de raccordement
5. Chambre à Projection Temporelle (TPC)
6. Calorimètre Electromagnétique (ECAL)
7. Solénoïde supraconducteur
8. Calorimètre Hadronique (HCAL)
9. Chambre à Muons
10. Calibration Laser

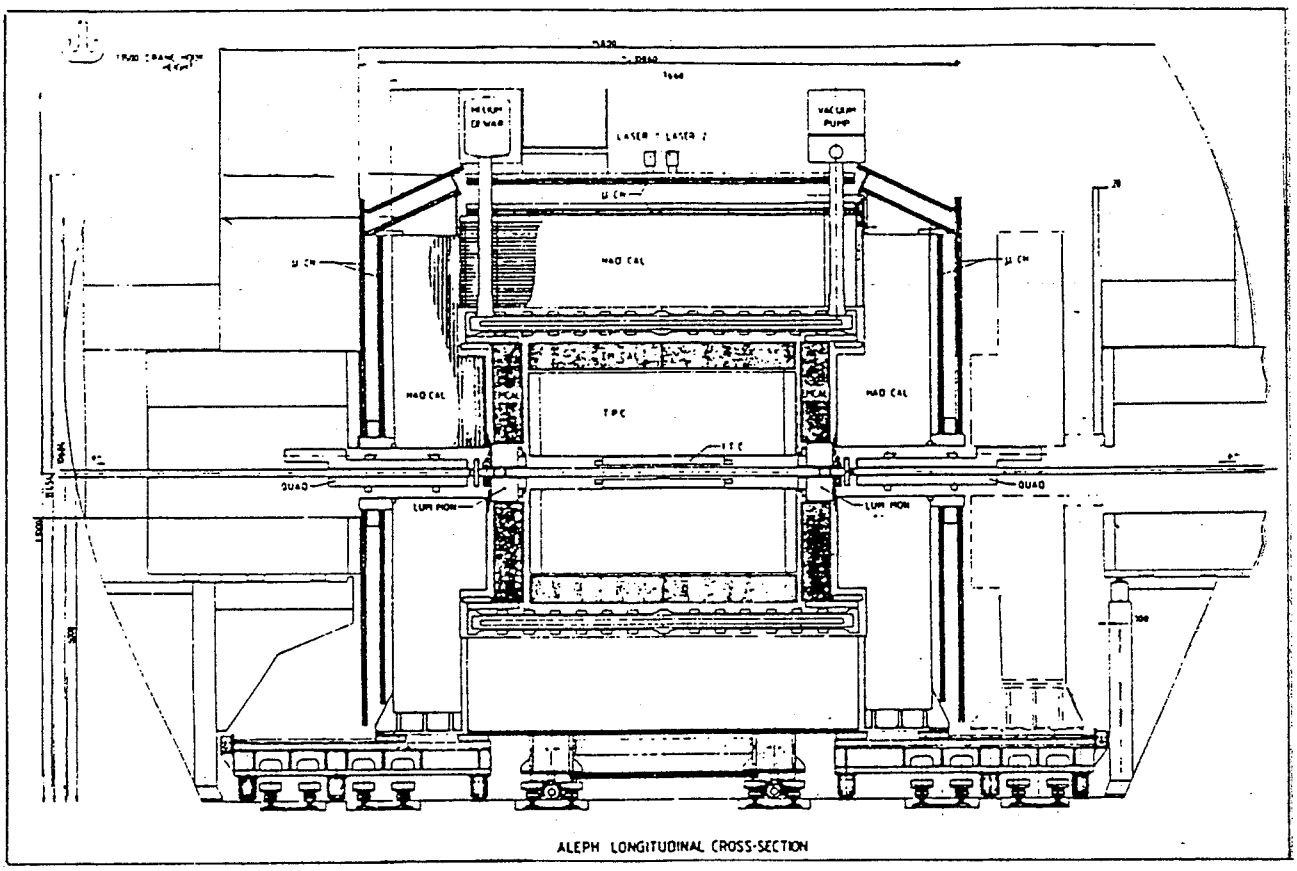


Fig. 2 Vue longitudinale d'ALEPH

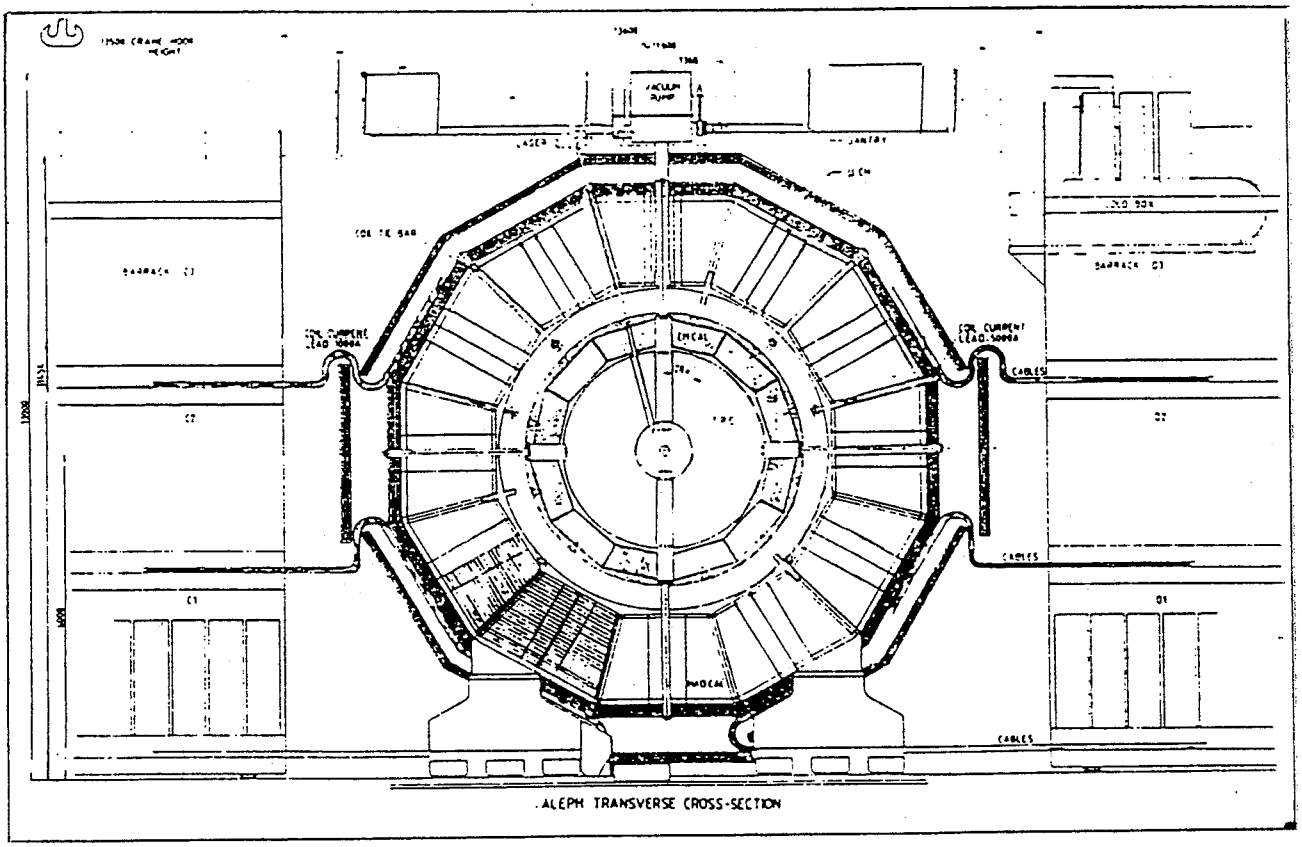


Fig. 3 Vue transversale d'ALEPH

Le 4 couches intérieures, y compris ECAL sont soumises à un champ magnétique de 1,5 Tesla par un solénoïde supraconducteur de 6m de diamètre pour 7m de long qui est parcouru par un courant de 5000 Ampères.

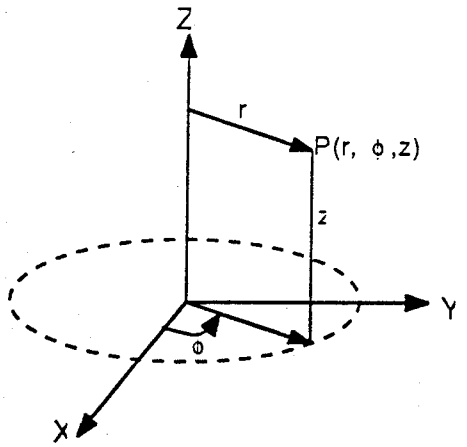
Des moniteurs de luminosité sont placés symétriquement de chaque côté du point de collision et couvrent une région angulaire de 2° à 7° autour du faisceau. Ils sont sensibles aux électrons des événements de type Bhabha à petits angles. Ils sont formés de plans de tubes à dérive suivis de détecteurs de gerbes similaires à ceux du calorimètre électromagnétique. Leur résolution est $0.20/\sqrt{E}$.

Les détecteurs de traces

Le détecteur deVertex, l'ITC et la TPC permettent de mesurer les trajectoires des particules chargées en localisant l'ionisation laissée dans le matériau du détecteur au passage des particules.

Le détecteur de Vertex [9] est composé de très fines bandes de silicium qui mesurent la position de la trace. Son rôle est d'améliorer de quelque 20% la précision de mesure sur l'impulsion des particules, et d'identifier des particules à courtes durées de vies (environ 10^{-12}s). La précision est de 10 μm en r-ϕ et de 20 μ en r-z. La séparation entre deux traçes est de 200 μm en r-ϕ et de 400 μm en r-z.

L'ITC [9] est composée de huit couches cylindriques concentriques de chambres à dérive. Elle couvre 97% de 4π et contient 960 fils sensibles. L'ITC assure en grande partie le dispositif de déclenchement pour les traces chargées et complète l'information de la TPC en améliorant le pouvoir de séparation de deux traces avec une précision de 100 μm, principalement dans le plan(r,ϕ).



Coordonnées cylindriques

La TPC [9] est le détecteur de traces le plus grand et le plus important. Il s'agit un cylindre de 3.6m de diamètre, rempli d'un gaz où les électrons,

créés au passage des particules chargées, dérivent dans un champ électrique vers l'une des deux plaques d'extrémité, chacune dotée de 20500 plaquettes qui collectent la charge déposée par ces électrons. La position d'une trace perpendiculairement au faisceau se mesure en comparant la valeur des charges recueillies sur des plaquettes adjacentes; sa position le long du faisceau est donnée par le temps de dérive des électrons jusqu'aux plaques d'extrémités. Les traces sont courbées par le champ magnétique de 15000 Gauss ce qui permet la mesure des impulsions des particules. La précision de cette mesure est d'environ 1% pour des particules de 10 GeV/C d'impulsion.

Les calorimètres

Les particules neutres ne laissent pas des traces dans les trois détecteurs internes. Leur énergie est mesurée dans deux calorimètres ECAL, HCAL. Les calorimètres sont formés alternativement de plaques d'une matière dense et de compteurs proportionnels. Les particules, neutres aussi bien que chargées, à l'exception des neutrinos et des muons, interagissent dans la matière en produisant des particules secondaires qui à leur tour interagissent, et ceci jusqu'à ce que l'énergie soit dissipée dans l'ionisation produite par les particules. Un échantillonage de cette ionisation est mesuré par les détecteurs de façon à pouvoir en déduire l'énergie initiale.

Le calorimètre ECAL [9] a eté conçu pour mesurer l'énergie des photons et des électrons. Il est divisé en 70000 secteurs qui pointent vers l'origine de la collision e^+e^-. Chaque secteur est séparé en profondeur en trois compartiments qui donnent une carte détaillée du dépôt d'énergie et permettent aussi l'identification des particules électromagnétiques. Des particules hadroniques peuvent également déposer une partie de leur énergie dans ECAL. Le dépôt dans les trois couches dépendra de la nature de la particule incidente : les électrons déposent environ 22% de leur énergie dans le premier niveau, 72% dans le deuxième et 8% dans le troisième. A 10 GeV, 20% des pions n'interagissent pas dans le calorimètre. Ils déposent en général moins de 70% de leur énergie dans le troisième niveau. ECAL a une résolution d'environ $0.18/\sqrt{E}$.

Le calorimètre HCAL [9] a été conçu pour mesurer l'énergie et la direction des hadrons. Il est constitué de multiples couches de chambre à fils qui échantillonnent l'énergie déposée dans du fer par les particules

ionisantes, avec une résolution de $0.8/\sqrt{E}$. HCAL donne deux types d'information : l'énergie déposée (signal des tours) et les déclenchements de fils. Ceux-ci donnent la trajectoire détaillée des particules permettant la discrimination entre Pions et Muons, les informations s'ajoutent à celles de deux chambres servant à l'identification des Muons. 99% des hadrons sont arrêtés après passage dans les deux calorimètres.

Mesure des muons

Les muons sont les seules particules chargées qui puissent traverser l'épaisseur de fer de 1.2m de la culasse de l'aimant et atteindre la chambre à muons. Cette dernière est constituée de deux plans de chambres à fils. La séparation des deux couches est de 50 cm, ce qui permet de mesurer la direction de traces avec une précision de 10-15 mrad.

1.2 Acquisition et traitement de données

ALEPH utilise largement des derniers développements de l'électronique et des techniques de calcul. Les données sont recueillies dans 500000 canaux électroniques.

L'acquisition de données d'ALEPH [9], utilisant FASTBUS, est un système hiérarchisé (voir Fig.4). Elle comprend :

- Les **électroniques de lecture** (Front-end electronics) qui sont intégrées dans les détecteurs.

- Le **timing, le système de déclenchement** (trigger et MTS) synchronisent les électroniques de lecture et informent les contrôleurs de lecture sur la disponibilité de données.

- Les **contrôleur de lecture** (ROC) initialisent les modules électroniques, acquièrent les informations, structurent les données et effectuent la première calibration.

- Les **EBs** (Event Builder) construisent l'événement au niveau de chaque sous-détecteur, effectuent les formatages de données et les transmettent au MEB (Main Event Builder).

- **L'ordinateur hôte** collectionne toutes les données venues du MEB et des ordinateurs des sous-détecteurs, effectue l'analyse d'événements en ligne et stocke les événements sur disque pour l'analyse en différé.

Les processeurs, intégrés à l'équipement, sont les microprocesseurs de la famille 68000 avec système d'exploitation OS-9. Le composant principal du système d'acquisition est un cluster VAX qui se compose de :

- 1 VAX 8700 : l'ordinateur principal
- 1 VAX 8250 : pour le détecteur TPC
- 2 VAX 8200 : pour les détecteurs ECAL et HCAL respectivement
- une vingtaine de stations du travail de type VAX

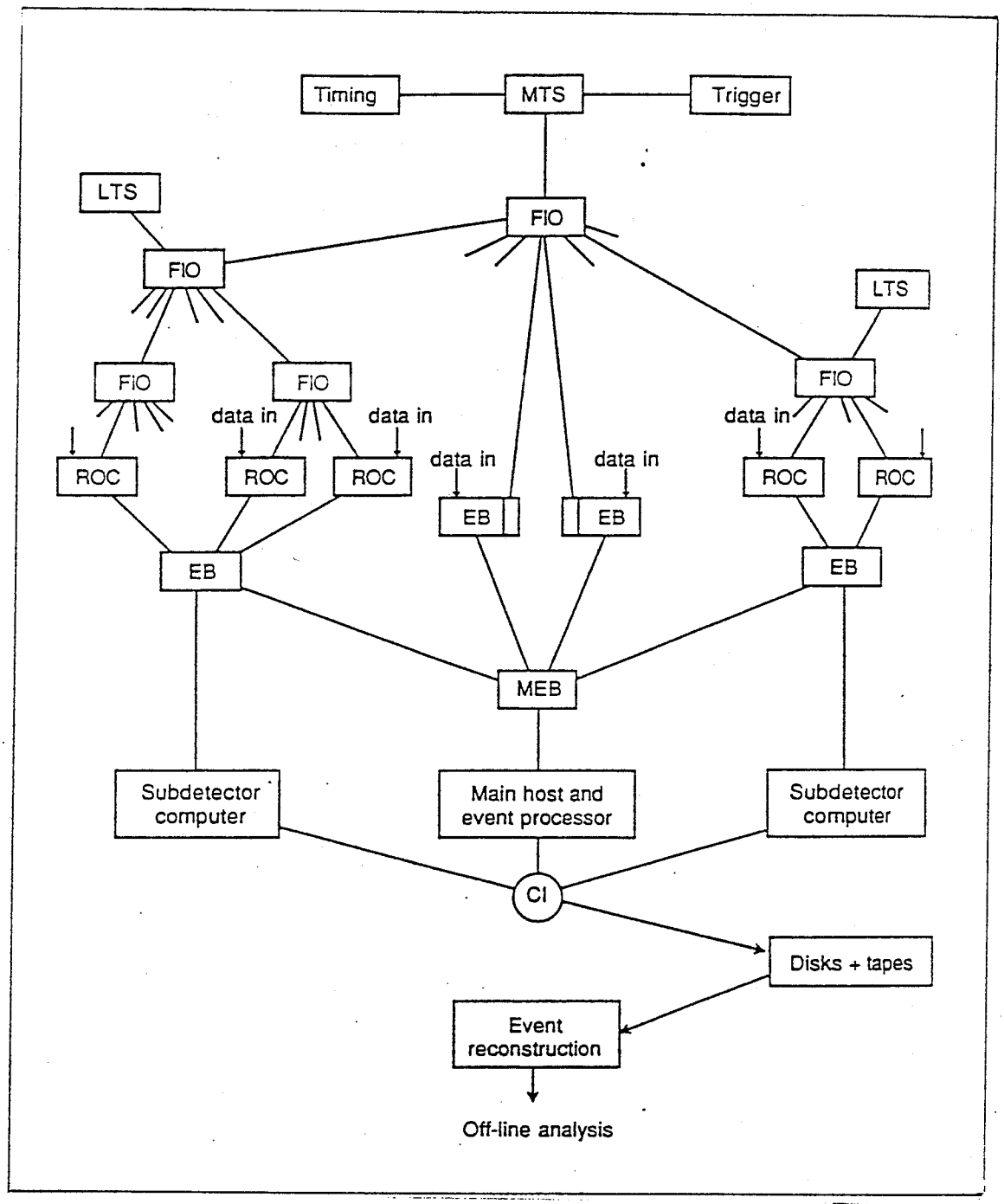


Fig. 4 Structure hiérarchisée de l'acquisition de données d'ALEPH, le flux des données présenté du haut vers le bas :

- Timing et le système de déclenchement (Trigger, MTS-Main Trigger Supervisor) synchronisent les électroniques de lecture
- ROC (Readout Controller) initialise les modules électroniques, acquiert les informations
- EB (Event Builder) construit l'événement au niveau de chaque sous-détecteur
  MEB (Main Event Builder) collectionne les données d'un événement venues de différents EBs
- L'ordinateur hôte collectionne, analyse et stocke les données pour le traitement en différé

Les données brutes, une fois recueillies par les ordinateurs d'acquisition, sont transmises au FALCON (Facility for ALeph COmputing and Networking)[10], qui est le système principal de reconstruction de données, situé dans la zone d'expérience. La liaison entre système d'acquisition et FALCON est assurée par deux disques communs de 1 Gigaoctet.

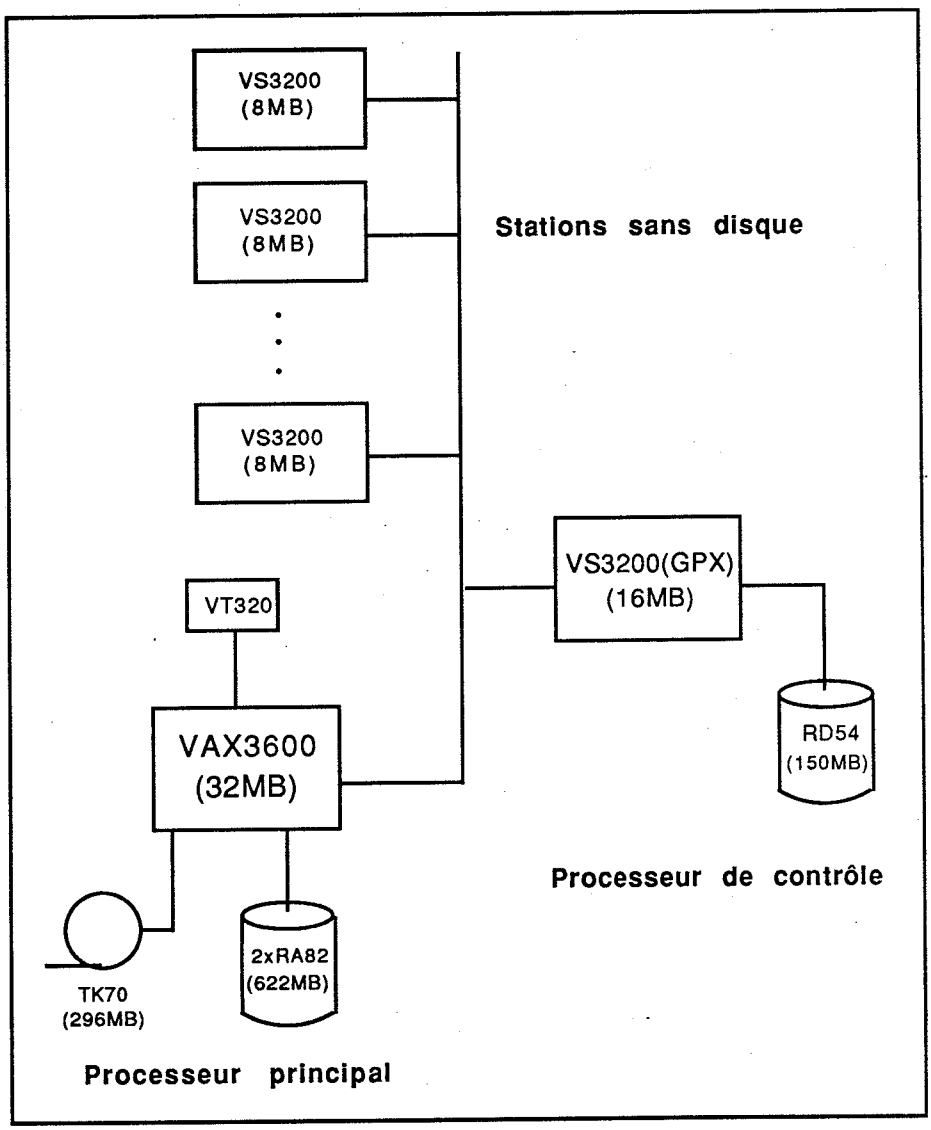


Fig. 5 La configuration du système principal de la reconstruction de données d'ALEPH (FALCON) : les reconstructions s'effectuent en parallèle dans un ensemble de stations DEC VS3200, contrôlées par une VS3200 à écran-couleur.

Pour des raisons de coût et d'efficacité, la reconstruction d'événements d'ALEPH se fait en parallèle : plusieurs événements sont reconstruits en même temps. Le système FALCON (Fig.5) est basé sur une architecture LAVC (DEC Local Area VAX Cluster) composée d'un VAX 3600 équipé de deux disques RA82 de 622 Mégaoctets et d'un ensemble de stations DEC VS3200 (8Mégaoctets de mémoire chacune). Un terminal VS3200 à écran-couleur sert de moniteur de contrôle. Avec une telle structure, ALEPH a obtenu 8 Mégaflops de puissance de calcul pour un flux d'entrée de 100 Kilooctet/s (données brutes) et un flux de sortie de 20 Kilooctet/s (données reconstruites).

Chapitre 2

Traitement de données

Dans son fonctionnement nominal, le LEP devrait permettre de collecter plusieurs millions de Z^0 par an. Une procédure de réduction de données doit être effectuée pour les condenser progressivement jusqu'à quelques chiffres qui confirment ou infirment ces prédictions théoriques du modèle standard.

La chaîne de traitement de données dans ALEPH est schématisée en Fig. 6. Les données brutes sont acquises par le système en ligne, puis traitées par le programme de reconstruction. Les résultats du programme de reconstruction constituent le POT (Production Output Tape), qui est le point de départ de l'analyse physique. A partir du POT, on sélectionne les événements présentant des caractéristiques particulières, calcule les nouvelles quantités physiques et prépare les histogrammes de ces quantités.

De nombreux logiciels ont été développés pour effectuer ce travail. Dans ce chapitre seront présentés les méthodes principales de conception de logiciels utilisés dans ALEPH et quelques logiciels utilisés pour le traitement de données. La plupart d'entre eux ont été développés par les membres d'ALEPH.

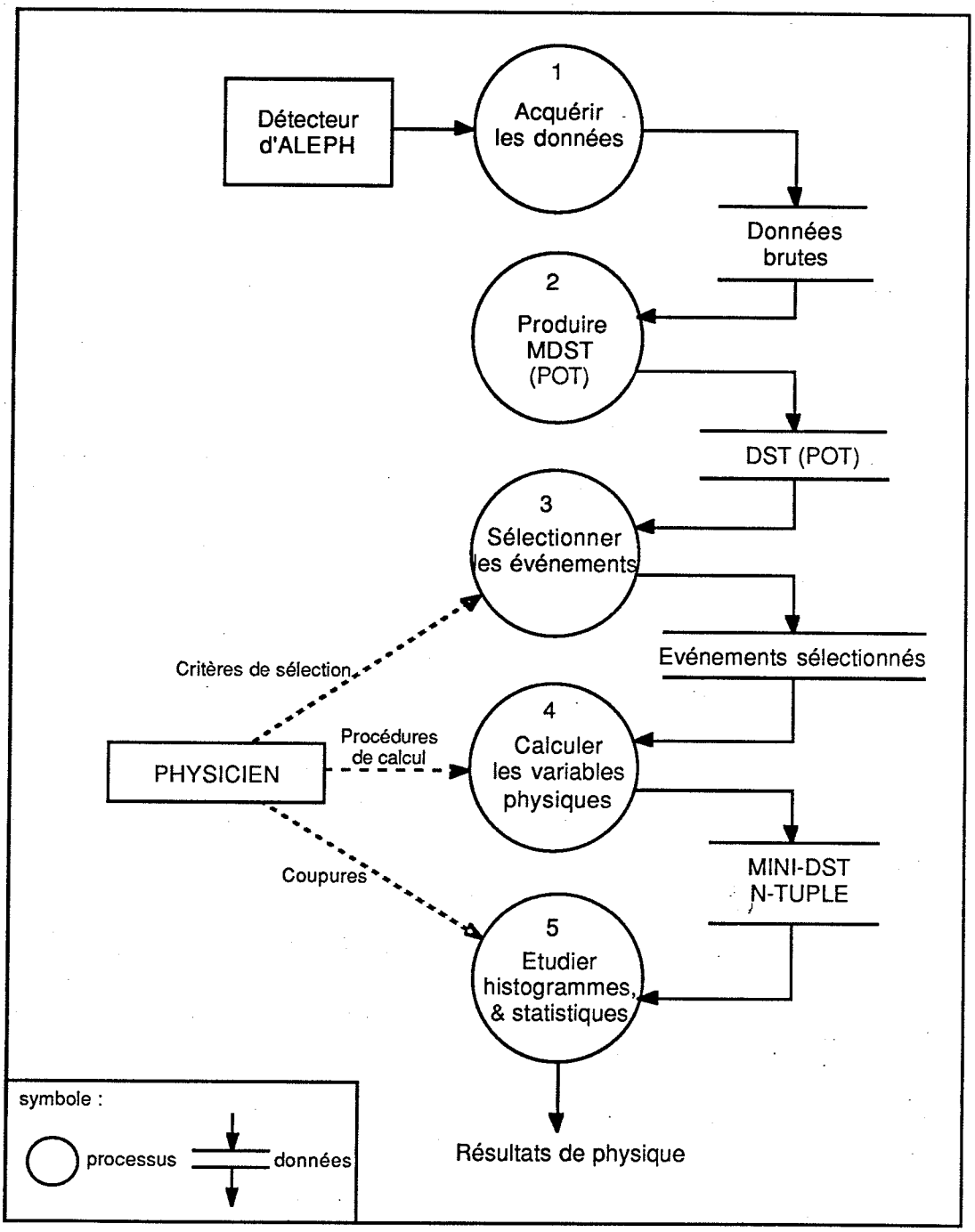


Fig. 6 Traitement de données : les données sont sélectionnées progressivement par différents processus sous contrôle de physiciens

2.1 Génie logiciel et organisation de données

Les logiciels utilisés dans le domaine de la physique des particules, comme tous les logiciels, ont leur propre cycle de vie - période allant de la décision initiale d'implémentation d'un logiciel à la fin de son utilisation. Cette période est composée de plusieurs phases :

- Demande des utilisateurs
- Analyse
- Conception
- Implémentation et vérification
- Opération et évolution

De nombreuses méthodes de génie logiciel sont utilisées pendant cette période pour

- accélérer le processus de production de logiciel
- améliorer la qualité du système
- construire une application qui satisfait précisement à la requête des utilisateurs

ALEPH a choisi **SASD** (Structured Analysis and Structured Design) [11,12,13] et le **modèle Entité-Association** (Entity-Relationship model) [13,14] comme méthodes principales pour la conception de logiciels. SASD est une collection de techniques pour modéliser le système réel. La fonction essentielle de SASD est de diviser un système en sous-ensembles plus petits et plus simples.

La phase d'analyse **SA** concentre l'intérêt sur le **processus** et le flux de données (Process modelling). La représentation graphique de ce modèle est le DFD (Data Flow Diagram). Dans le DFD, le système réel se présente comme un réseau de processus qui transforment les données. Chaque processus peut être à son tour décrit par un nouveau DFD plus détaillé. Le flux de données est décomposé en même temps. La Fig.6 est un exemple de DFD.

Dans la phase **SD**, le système réel est modélisé dans le diagramme structuré (SC - Structure Charts) qui se compose de différents modules de logiciel (par ex. fonction et subroutine en FORTRAN). Le flux de contrôle

et le flux de données entre modules sont aussi indiqués.

Le **modèle Entité-Association** est une stratégie pour définir la structure de données et, comme tout modèle de données, consiste en :

- des structures de données bien définies,
- une collection d'opérateurs pour manipuler les données,
- des procédures de validation pour vérifier que les données obéissent aux contraintes imposées.

Dans le logiciel d'ALEPH on emploie le modèle ER à l'aide du système ADAMO (Aleph DAta MOdel) [15,16,17]. Les données sont structurées sous forme d'ensembles d'ENTITES définies par les valeurs de propriétés dites ATTRIBUTS. Les RELATIONS (fonctions) entre ensembles sont aussi définies.

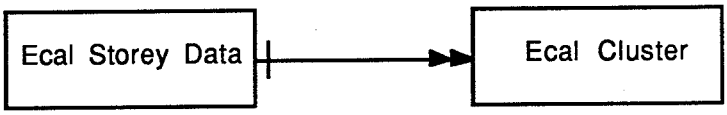
Le modèle ER se prête bien à une représentation par diagrammes, les ERDs (Entity-Relationship Diagram), composés de boîtes (les ensembles) et de flèches (les relations). On utilise par la suite un langage de définition de données ou DDL (Data Definition Language) pour décrire en détail chaque ensemble d'entités, ses attributs et ses relations.

On trouvera Fig.7 un exemple de ERD et son DDL correspondant. C'est un extrait de la description de la reconstruction des énergies des gerbes dans le Calorimètre électromagnétique. L'exemple complet de cette description se trouve dans l'Annexe 1.

Le système ADAMO vérifie que la définition des données soit complète et cohérente, et la stocke dans un dictionnaire, à partir duquel on produit des parties de code qui matérialisent les données dans le cadre de différents langages de programmation: FORTRAN, C, PROLOG, SQL, etc. La Fig.11 montre un exemple de traduction du DDL en PROLOG utilisée dans le système PIGAL

Le DDL, référence unique sur les structures de données d'ALEPH, garantit la consistance et la validité des données à travers toutes les phases du cycle de vie du logiciel. Une grande partie des systèmes logiciels de l'expérience utilise le modèle sous différentes formes.

1) ERD



Dans ERD, les boîtes représentent des ENTITE, les flèches représentent des RELATION, la double flèche signifie que un "Ecal Cluster" contient plusieurs "Ecal Storey Data", la barre sur la flèche signifie que certains "Ecal Storey Data" n'appartiennent à aucun "Ecal cluster".

2) DDL

```
DEFINE ESET

ESDA : 'Ec storey data'
   = (ThetaJ       = INTE [1,228]    : 'J (theta) index',
      FhiI         = INTE [1,384]    : 'I (phi) index',
      Depthk       = INTE [1,3]      : 'Stack number',
      MEnergy      = REAL [-1.,90.]  : 'Energy in Gev',
      SubComponent= INTE [1.5]       : 'subcomponent number',
      IOaddress    = INTE [1,221184]: 'Add:encoded i,j,k address',
      DIrection    = INTE [1,25000]  : 'Ndeb:start add in roseve',
      EcalRegion   = INTE [1,36]     : 'Region number')
   ;

ECLU : 'Ecal cluster'
   = (Charge,
      Energy,
      ...)
   ;

END ESET

DEFINE RSET

  (ESDA [0,1] -> [1,*] ECLU)
     : 'Clusters are constructed from Storeys';

END REST
```

Fig. 7 Les exemples de ERD (Entity-Relationship Diagram) et DDL (Data Definition Language) : ils sont tirés du document DDL 'Calorimètre Electromagnétique'.

2.2 Logiciels principaux pour le traitement de données d'ALEPH

Les logiciels principaux utilisés dans la chaîne du traitement de données d'ALEPH sont le logiciel de reconstruction (JULIA), le logiciel d'analyse physique (ALPHA) et le logiciel d'analyse statistique (PAW).

La physique que l'on veut étudier dans un détecteur comme ALEPH est caractérisée par les leptons, quarks et gluons émis à la collision. Les lepton chargés sont détectables. Les quarks et les gluons n'existent pas à l'état libre mais forment les hadrons, qui sont composés de plusieurs quarks. Ces dernières particules sont reconstruites par le programme JULIA. Les résultats de la reconstruction permettent d'analyser, en utilisant le logiciel ALPHA, les événements et d'étudier la physique associée à la recherche d'états finaux particuliers comme par exemple la physique des leptons, la physique associée aux jets ou encore la physique associée aux particules manquantes avec la différence entre l'énergie reconstruite et l'énergie de collision...

On trouve Fig.6 les positions de ces logiciels dans la chaîne d'analyse :

- **JULIA** correspond au processus 2. Il produit le POT à partir des données brutes.
- **ALPHA** correspond aux processus 3 et 4. Il effectue l'analyse physique en utilisant le POT comme données d'entrée et produit les histogramme ou les n-tuples.
- **PAW**, un système d'analyse statistique de données, correspond au processus 5. Il est la dernière étape de la chaîne de traitement.

JULIA - Logiciel de reconstruction

JULIA reconstruit à l'aide de l'information, généralement mesure de signaux électriques des différents détecteurs, les caractéristiques des particules qui ont été produites lors de la collision.

Il effectue les associations topologiques suivantes :

- association des coordonnées des impacts mesurés dans les ITC et TPC, selon des trajectoires théoriques (hélices)
- regroupement des cellules calorimétriques voisines, ainsi qu'une première identification de particules (électrons, muons, hadrons chargés ou neutres, photons gamma)

Reconstruction dans la TPC [18] :

La TPC donne une image, point par point, des traces chargées. La reconstruction des traces se fait par les étapes suivantes :

- relier les points dans l'espace, former les traces
- calculer leurs impulsions
- réduire l'information des fils en une estimation de temps et de charge
- associer les fils et traces TPC
- mesurer la densité d'ionisation pour chaque trace dont on déduit une hypothèse d'identification (choisie parmi les quatres possibilités: e, π, K, p).

Reconstruction dans les ECAL, HCAL [19,20,21] :

La mesure de l'énergie et l'identification des particules dans ECAL et HCAL se font aussi en plusieurs étapes :

- regrouper un ensemble de cellules touchées voisines en amas qui sont considérés comme entités de base pour une reconnaissance de forme ou pour une association éventuelle avec une particule chargée vue dans la TPC
- associer ces amas avec les traces de la TPC, permettant de distinguer deux types d'amas : chargés ou neutres
- associer les amas de ECAL et HCAL
- effectuer une reconnaissance de forme (par la répartition en étages, point de départ et de fin en profondeur de la gerbe) et déterminer la nature de la particule ou la présence dans un même amas de deux particules différentes.

Le programme est écrit en FORTRAN 77 standard. Il utilise BOS [22]

comme gestionnaire de mémoire ainsi que pour les opérations d'entrée-sortie sur bande magnétique, ou sur disque (accès direct). JULIA contient plus que 70000 lignes de code et occupe environ 5 Mégaoctets de mémoire.

ALPHA - Logiciel d'analyse physique

ALPHA (ALeph PHysics Analysis package) [23] est un logiciel d'analyse physique pour les données d'ALEPH. Il offre quelques facilités permettant de simplifier la programmation pour l'analyse.

ALPHA utilise en entrée un fichier dit **répertoire d'événement** (utilisé par la plupart des logiciels d'analyse d'ALEPH) pour accélérer l'accès aux données.

Les fonctions principales d'ALPHA sont les suivantes :

- assurer la lecture des événements par un accès direct aux données et effectuer les sélections selon les critères physiques définis par l'utilisateur.
- convertir les données reconstruites dans des structures internes en facilitant la manipulation.
- fournir une bibliothèque de subroutines de physique : calcul de nouvelles quantités, accès aux contenus de certaines variables, manipulation des 4-vecteurs d'impulsion, calcul de la topologie des événements : valeurs propres, vecteurs propres des tenseurs, sphéricité, thrust, masse invariante de deux particules, recherche des jets, etc.

Les sorties d'ALPHA sont variées : répertoires d'événements, événements complets, histogrammes ou N-Tuples utilisés par les logiciels d'analyse statistique, par exemple PAW.

PAW - Logiciel de représentation statistique de données

PAW (Physics Analysis Workstation) [24] est un outil général d'analyse et de présentation de données. Il permet d'effectuer l'analyse statistique sur les objets familiers aux physiciens, par ex. histogramme, n-tuples, vecteur, etc.

PAW est interfacé avec de nombreux types de station de travail graphique, permettant d'obtenir une représentation d'histogramme de haute qualité.

Les fonctions essentielles de PAW sont les suivantes :

- accéder aux données sous forme d'histogramme, n-tuples ou vecteur
- créer, remplir et manipuler les histogrammes
- représenter les histogrammes en graphique 2D ou 3D
- produire des graphiques prêts pour publication

Ce logiciel est écrit entièrement en FORTRAN 77 et contient plus de 100000 lignes de code.

2.3 Nécessité d'un environnement interactif et graphique

L'analyse de données est un processus de réduction des données sous contrôle d'un physicien (Fig.6). Normalement, chaque physicien doit faire ces travaux **"en batch"** (traitement différé). Il doit alors envisager toutes les anormalies qui pourraient apparaître pendant le traitement. Les inconvénients sont multiples: il est difficile de tout prévoir et de plus il existe différentes façons de traiter les anormalies suivant le contexte. Mais quand on travaille "en batch", aucune intervention n'est possible durant l'exécution.

Un environnement interactif et graphique serait nécessaire pour différentes phases du travail :

- Pendant le développement du programme, on a besoin de traiter peu d'événements, ce qui permet de tester les étapes intermédiaires des algorithmes. On a besoin de produire et de représenter des histogrammes sur des variables sensibles, de changer paramètres et algorithmes d'une manière interactive, de ré-exécuter le programme pour tester l'effet de ces changements.

- Pendant l'analyse, on a besoin d'accéder rapidement et facilement aux événements sélectionnés, de produire des histogrammes, de visualiser des événements intéressants, de vérifier les données de ces événements, etc.

- Pendant ces deux phases on a besoin d'un moyen de **visualiser** des données d'ALEPH, leur structure, leur relation ...

Depuis 5 ans, l'utilisation de moyen interactif et graphique dans la physique de particules est devenue possible grâce au dernier développement de stations de travail graphique en réseau. Dans ALEPH ainsi que dans d'autres expériences, différents systèmes sont développés. En voici quelque exemples :

PAW : (Physic Analysis Workstation) un système de représentation statistique des données développé an CERN [24]

IDA : (Interactive environnement for Data Analysis) un système interactif développé à SLAC [25]

KAL : (Kinematic Analysis Language) un système interactif d'analyse développé à ARGUS

TIP : (Table Interaction and Plotting) interface interactive à ADAMO basée sur PAW [26]

DALI : système interactif graphique 2D pour ALEPH [27]

REASON: (Realtime Event Analysis Workstation Project) un nouveau système développé à SLAC, utilisé sur une machine NeXT [53]

PIGAL : (Prolog Interactive and Graphics for Aleph analysis) un logiciel d'analyse interactive de données pour ALEPH

Ces logiciels intègrent plus ou moins les parties logiques suivantes :

1 **L'interface d'utilisateur** qui analyse les demandes de l'utilisateur sous différentes formes, par ex. langage de commande (naturel ou programmation), menu, pictogramme ...
2 **Module de contrôle** qui synchronise différents composants du système effectuant chacun une fonction spécifique
3 **Serveur de données** qui accède aux événements
4 **Sélection de données** permettant de choisir les données qui correspondent aux critères physiques
5 **Calcul de nouvelles quantités physiques** : codage d'algorithmes
6 **Préparation d'histogrammes** qui crée, remplit, manipule les

histogrammes
7 **Préparation de données pour la visualisation d'événements**
8 **Préparation de données pour la visualisation des contenus et des structures de données**
9 **Représentation d'histogrammes**
10 **Visualisation d'événements en graphique 2D ou 3D**
11 **Représentation de données avec leur structure**
12 **Accès à la base de données**

Pourtant ils sont tous différents au point de vue

- de la structure du système
- de l'organisation des données
- de la réalisation de l'interface avec l'utilisateur
- du niveau d'utilisation de graphique 2D, 3D (matériel, logiciel)
- des utilitaires pour l'aide à l'analyse physique
- du niveau de manipulation d'histogrammes
- de leur rôle dans la chaîne de traitement de données

Dans le chapitre suivant sera présenté l'un de ces logiciels - PIGAL, développé au CPPM, Marseille, dans le but d'unifier d'accès aux données et leurs manipulations à tous les niveaux de la chaîne d'analyse.

Chapitre 3

PIGAL, un programme interactif de visualisation et d'analyse

Le Projet PIGAL (Prolog Interactive and Graphic for ALEPH analysis), lancé au début de 1988, a visé à réaliser un environnement multi-fonctionnel permettant à l'utilisateur d'effectuer les opérations traditionnelles d'analyse de données d'une manière plus naturelle. Désormais un tel environnement est mis en service dans ALEPH, la majorité de fonctionnalités décrites dans les spécifications du projet ont été réalisées.

Dans ce chapitre, les spécifications initiales du projet seront résumées, suivies de la description de la structure actuelle du système et de ses fonctionnalités les plus significatives.

3.1 Spécifications initiales de PIGAL

Les spécifications sont divisées en 3 parties :

- Interface utilisateur
- Fonctionnalités pour l'analyse
- Modularisation en processus

Interface avec l'utilisateur

- **Menu** (pour l'utilisateur non expert) :
  Le menu est un moyen simple de familialisation avec le système, qui doit couvrir la plupart des besoins.

- **Langage de commande** (pour l'utilisateur expert) :
  Le langage de commande doit être le plus naturel possible et capable d'exprimer les opérations essentielles de sélection de données structurées dans le modèle Entité-Association.

- **Macros** :
  Une macro est un groupe de commandes utilisées fréquemment, avec un nom spécifique ("nickname") invoqué par l'utilisateur pendant l'analyse. Elle pourrait être publique, contenue dans PIGAL pour une utilisation générale; ou bien privée, créée et utilisée par l'utilisateur. L'utilisateur peut créer et enregistrer une macro soit pendant une session interactive, soit en dehors de la session à l'aide d'un éditeur.

- Possibilité d'enregistrer dans un **journal** (log file) les commandes utilisées pendant la session interactive pour une utilisation ultérieure

- Possibilité d'obtenir les **informations d'aide** (on-line help)

Fonctionnalités pour l'analyse

- Facilité d'**accéder à différents types de fichiers** : données brutes, POT, Mini-DST

- Capacité d'**accéder aux données** d'ALEPH sans demander une connaissance détaillée à priori sur la structure de données, en particulier le changement de DDL n'affecte pas l'utilisateur

- Capacité de **sélectionner** les événements

- Capacité de **visualiser** les événements en graphique 3D

- Capacité de **modifier interactivement les structures d'entités**, ou bien d'ajouter dynamiquement de nouvelles entités

- Manipulation et utilisation avancée d'**histogrammes**, par ex. l'utilisateur demande une représentation d'un histogramme, ensuite il pointe avec la souris sur l'écran un canal intéressant et obtient automatiquement la liste des événements correspondants

- Capacité de soumettre automatiquement **le traitement en différé** (batch job) avec les critères physiques déterminés au préalable dans une session interactive

(Les trois dernières fonctionnalités ne sont pas prêtes dans la version présente du programme.)

Modularisation en processus

l'interface utilisateur est réalisée en langage PROLOG. La plupart de fonctions sont réalisées en FORTRAN. Les modules nécessaires sont les suivants :

- modules pour la **réduction de données** : logiciel de reconstruction JULIA, facilité pour accéder aux fichier d'événements et pour enregistrer les événements après une sélection, algorithme de sélections
- module de **visualisation d'événements**
- module de **manipulation de structure de données**
- module de **manipulation et représentation d'histogramme**
- module de **l'analyse physique** contenant les subroutines standards de calcul de quantités physiques

3.2 Structure actuelle du logiciel PIGAL[28]

Le logiciel PIGAL est basé sur une architecture modulaire, chaque module ayant une fonction spécifique. La communication entre modules se réalise à travers les structures de données de BOS. Dans cette section, l'explication porte sur :

- la structure actuelle du système
- la justification de l'emploi de PROLOG pour réaliser l'analyseur

grammatical
- la base de connaissances
- l'interface PROLOG-FORTRAN développée pour PIGAL.

Toutes les commandes et macros invoquées comme exemple seront décrites individuellement. Leurs spécifications détaillées se trouvent dans le guide d'utilisation [29]. Les termes "table" et "colonne" seront employés parfois au lieu de "entité" et "attribut".

Organisation du système

Comme la plupart de systèmes d'analyse, PIGAL se compose de trois parties principales (voir la Fig.8):

Interface utilisateur

Pour en faciliter l'utilisation, on a choisi un langage quasi naturel comme base de l'interface utilisateur. Un analyseur grammatical en PROLOG digère les demandes de l'utilisateur avant leur exécution.

Support des données d'événement

- Les données utilisées dans PIGAL sont lues sur les fichiers d'événements d'ALEPH, à différents niveaux de traitement : données brutes issues des détecteurs, données reconstruites issues du programme de reconstruction et ultérieurement Mini DST

- PIGAL peut reconstruire certaines entités annexes (ex. **esda**) dynamiquement si la commande les concerne. Les subroutines nécessaires du programme de reconstruction (JULIA) sont intégrées dans PIGAL et appelées suivant les besoins.

- Un utilitaire SELTAB effectue les sélections éventuelles sur les entités concernées

Action

C'est un ensemble des fonctionnalités se réalisant soit en pur PROLOG soit par coopération de PROLOG et de FORTRAN. Cette partie contient la plupart des utilitaires, elle intègre aussi les sous-systèmes développés dans des contextes différents.

Le diagramme de flux de données (DFD) de PIGAL se trouve dans l'Annexe 2.

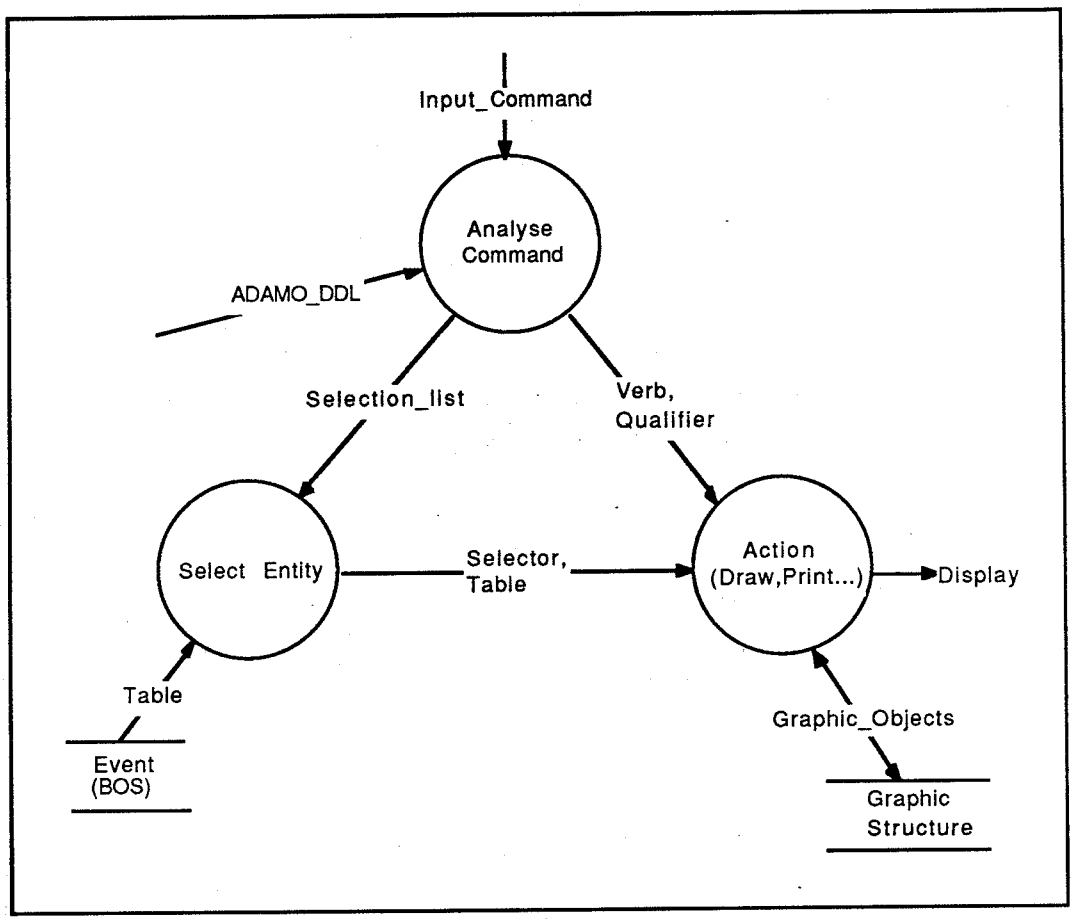


Fig. 8 L'architecture de PIGAL contient 3 parties :
1) L'interface d'utilisateur (Analyse Command) : Cette partie analyse les commandes à l'aide de ADAMO DDL, contrôle la séquence des sélections effectuées par le support de données
2) Le support de données (Select Entity) : Cette partie lit les événements d'un fichier, effectue les sélections, crée le sélecteur
3) Les modules de fonctionnalités (Action) : Cette partie reçoit le sélecteur et les données, effectue les fonctions (Draw, Print, Histogram ...) à l'aide des structures de données graphiques. Elle contient aussi certains sous-systèmes développés dans des contextes différents

Justification de l'emploi de PROLOG pour réaliser l'analyseur grammatical

Le langage de commande de l'utilisateur est EARL (Entity and Relationship query Language) [30], un langage quasi naturel conçu pour exprimer les opérations essentielles dans la sélection des données ayant une structure Entité-Association. Le système permet au physicien de poser ses questions naturellement, par exemple

```
Draw tracks momentum>1 without associated calobjects.
Select events with 2 leptons.
```

La syntaxe du langage de commande est décrite Fig.9. Cette syntaxe permet d'effectuer une sélection de l'entité par des conditions portant sur plusieurs attributs ou par ses relations avec d'autres entités.

Un analyseur grammatical est nécessaire pour interpréter ce type du langage. Celui de PIGAL est écrit en langage PROLOG. Ce choix est basé sur le fait que PROLOG est un résultat de la recherche sur la compréhension du langage naturel et sur la programmation en logique, il s'adapte donc bien à EARL. L'analyseur de PIGAL est constitué d'un programme effectuant les déductions logiques permettant de transformer facilement une requête de l'utilisateur en une séquence d'appels à des subroutines FORTRAN propres à la satisfaire. Il utilise également la représentation de certaines connaissances utiles au travail d'analyse : structure des données, liste des fonctions disponibles, etc.

PROLOG offre une interface avec d'autres langages, ce qui permet d'utiliser certains programmes existants et de développer des algorithmes en FORTRAN et C. Ceci est largement utilisé dans PIGAL.

Certains éléments du langage PROLOG sont détaillés dans l'Annexe 3. Une connaissance minimale sur ce langage sera utile pour mieux comprendre la réalisation de PIGAL.

```
command          :=  action object;
action           :=  verb { "/" qualifiers};
qualifiers       :=  qualifying-unit;
qualifying-unit  :=  option  "="  value;
option           :=  colour;
                 |   view;
                 |   representation;
   . . .

object           :=  entity  selection;
selection        :=  none;
                 |   attribute comparison number {and selection};
                 |   connector object;
connector        :=  in;
                 |   with;
                 |   without;
                 |   out;
comparison       :=  "<";
                 |   "=";
                 |   ">";
verb             :=  draw;
                 |   print;
                 |   histogram;
                 |
   . . .
```

Fig. 9 La syntaxe de EARL utilisée dans PIGAL :
EARL (Entity and Relationship query Language) est un langage quasi naturel, conçu pour exprimer les opérations essentielles dans la sélection des données qui ont une structure Entité-Association

Base de connaissances

Quand PIGAL reçoit une commande de l'utilisateur :

```
draw calo energy>10
```

il cherche à identifier ce qu'est **"calo"** et ce que signifie **"energy>10"**. Il trouve que **calo** est une entité complexe formée de deux entités élémentaires **esda** et **hsda**. Il trouve aussi, par exemple, que ces deux entités ne sont pas instanciées et qu'il faut les reconstruire. Il trouve dans sa base de faits qu'il existe des subroutines dans le programme de reconstruction qui peuvent faire ce travail. Ensuite le programme

identifie une sélection sur l'attribut **energy** et trouve de la même manière que **energy** est un attribut de l'entité **calo.** La commande est ainsi validée et transformée en une série d'appels à des subroutines FORTRAN.

Cet exemple traverse presque toutes les parties de la base de connaissances :

- structure tabulaire des données
- structure graphique arborescente (entités complexes)
- description des modules de reconstruction
- relations entre représentation des entités physiques et subroutines FORTRAN
- description des modules de sélection

Les deux premières parties sont décrites ici, car ce sont les bases communes. Les autres seront expliquées dans les sections suivantes avec leur utilisations internes.

Structure de données - ALEPH DDL

Afin de pouvoir l'intégrer dans le programme PIGAL, le DDL est converti par l'un des outils d'ADAMO sous forme de faits PROLOG. La Fig.10 montre les deux représentations pour l'entité **esda.**

Structure arborescente - Entité complexe

Certaines entités élémentaires d'ALEPH ne peuvent pas être dessinées directement. Si l'on voulait effectuer quelque opération sur cette sorte d'entités, par exemple l'entité **eclu**, le programme doit utiliser les connaissances suivantes:

```
entity(esda) ->;
entity(eclu) ->;
complex-entity(eclu,with.nil,esda.nil,v) ->;
```

La dernière règle signifie que **eclu** est une entité complexe, si on demande de visualiser un **eclu** sélectionné, la représentation porte en fait sur l'entité **esda.**

ADAMO DDL

```
DEFINE ESET

ESDA
   : 'Ec storey data'
   = (ThetaJ       = INTE [1,228]    : 'J (theta) index',
      FhiI         = INTE [1,384]    : 'I (phi) index',
      Depthk       = INTE [1,3]      : 'Stack number',
      Energy       = REAL [-1.,90.]  : 'Energy in Gev',
      SubComponent= INTE [1.5]       : 'subcomponent number',
      IOaddress    = INTE [1,221184]: 'Add:encoded i,j,k address',
      DIrection    = INTE [1,25000]  : 'Ndeb:start add in roseve',
      EcalRegion   = INTE [1,36]     : 'Region number')
   ;

END ESET

DEFINE RSET

  (ESDA [0,1] -> [1,*] ECLU)
     : 'Clusters are constructed from Storeys';

  (ESDA [0,1] -> [0,1] ESDA)
     : 'index of the next storey of the same cluster';

END REST
```

Règles en PROLOG

```
esda(thetaj,1,1) ->;
esda(fhii,1,2) ->;
esda(depthk,1,3) ->;
esda(energy,2,4) ->;
esda(subcomponent,1,5) ->;
esda(ioaddress,1,6) ->;
esda(direction,1,7) ->;
esda(ecalregion,1,8) ->;
esda(eclu,3,9) ->;
esda(esda,3,10) ->;
```

Fig. 10 L'exemple de DDL
en haut : ADAMO DDL de l'entité ESDA (Ecal Storey Data)
en bas : le même DDL écrit en PROLOG, comme une partie de sa base de connaissances, utilisé dans PIGAL. Chaque règle contient : nom de l'entité, nom de l'attribut, type de l'attribut (entier, réel, relation) et numéro de l'attribut.

Interface PROLOG - FORTRAN pour PIGAL

PROLOG offre des possibilités de communication avec les langages traditionnels. Cette possibilité est capitale pour PIGAL, puisqu'il pilote des ensembles de logiciels écrits pour la représentation graphique 3D ainsi que des procédures préexistantes dans le programme de reconstruction et dans PAW. Il est exclu de les réécrire pour des raisons de compatibilité, ainsi qu'en raison de l'énorme investissement humain qu'ils représentent.

Il y a deux sortes d'interface PROLOG - FORTRAN utilisées dans PIGAL:

- Interface standard de PROLOG décrite dans l'Annexe 3
- Interface développée spécialement pour PIGAL

La différence entre les deux interfaces est présentée schématiquement Fig.11. Les deux assurent la même fonction, mais sont utilisées dans des contextes différents. L'interface standard est efficace si l'application contient peu d'appels. L'autre interface convient mieux quand on gère beaucoup de sous-programmes FORTRAN ou C. Dans PIGAL, PROLOG gère plus de 200 subroutines FORTRAN dont une vingtaine seulement par l'interface standard. On verra lors du chapitre suivant que ce partage est important pour l'architecture distribuée de PIGAL .

L'interface standard est simple, son inconvénient est que chaque subroutine a son propre identificateur, et chaque branche doit gérer proprement le transfert des arguments entre PROLOG et FORTRAN. C'est très difficile à utiliser dans une application comprenant des centaines d'appels différents. Pour résoudre ce problème on ajoute une couche supplémentaire de logiciel, qui permet l'appel de la subroutine depuis PROLOG, sous la forme :

```
fortran(nom de subroutine, liste d'arguments d'entrée,
liste d'arguments de sortie)
```

ici la liste de sortie est de la forme:

```
(type1.variable1).(type2.variable2)...nil
```

les types possibles sont : int (entier), real (réel), chxx (chaîne de caractères de longueur xx).

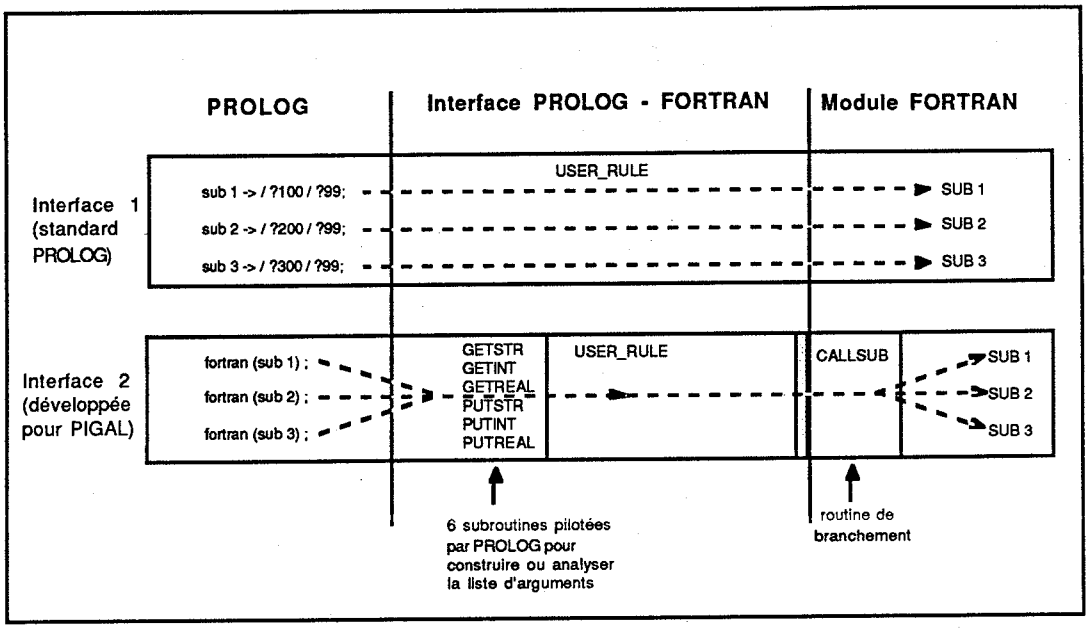


Fig. 11 Interface PROLOG-FORTRAN dans le contexte de PIGAL :

Interface 1 est fournie par l'interpréteur PROLOG, chaque appel dans PROLOG a une forme de prédicat externe qui est transformé dans l'interface en un appel FORTRAN

Interface 2 est développée pour PIGAL parce que l'utilisation de l'interface 1 devient malaise lorsque les subroutines externes sont très nombreuses. Ici plusieurs appels dans PROLOG sont transformés dans l'interface en un seul appel FORTRAN, le module FORTRAN gère le branchement vers l'adresse adéquate

Avec cette interface, on n'a besoin que de 7 subroutines pour gérer le passage PROLOG-FORTRAN (voir Fig.12). Six d'entre elles (getstr, getint, getreal, putstr, putint, putreal) servent comme passage d'arguments. La dernière (callsub) est responsable de tous les appels FORTRAN. Callsub fait appel à la fonction bibliothèque VAX : Lib$callg, pour effectuer le branchement vers l'adresse adéquate. Sur IBM, quelques lignes de code assembleur permettent d'obtenir le même résultat. L'adresse de chaque subroutine est obtenue par l'instruction external, qui fait correspondre à chaque subroutine une variable dont le contenu est l'adresse de la subroutine.

L'important dans ce mécanisme est qu'il fournisse une interface propre, unique, formelle entre les modules — une conditions nécessaire pour changer facilement le système vers une architecture distribuée plus tard.

Exemple d'appel

```
write("filename") -> fortran(write,"filename".nil);
```

Règles en PROLOG

```
fortran(s-ubroutine-name,i-nput-list) ->
     inarg putlist(i-nput-list)
     string-ident(x,s-ubroutine-name) callsub(x);
callsub(a) ->/?560/?99;
putstr(a,b) ->/?530/?99;
putint(a) ->/?540/?99;
putreal(a) ->/?550/?99;
getreal(a) ->/?570/?99;
getint(a) ->/?580/?99;
getstr(a) ->/?590/?99;
```

Subroutine user_rule

```
      subroutine user_rule(nb,error_found,error_nb)
      integer  nb,error_nb
      logical  error_found
      character*132  string,stringi,stringo,stringp
      error_found=.false.
      icase=nb/100
      goto(999,999,999,999,500),icase
500   continue
      ind=(nb-500)/10
      goto(999,999,530,540,550,560,570,580,590),ind
540   call get_integer(1,ival,error_found)
      call putint(ival)
      goto 999
560   call get_string(1,1,%ref(stringi),error_found)
      call callsub(string(1:1))
      goto 999
      call getint(ival)
580   call put_integer(1,ival,error_found)
      goto 999
         •
         •
      return
999   end

      subroutine callsub(name)
         •
         •
         call lib$callg(....)
         •
         •
      end

      subroutine putint(ival)
      subroutine putstr(string,lres)
      subroutine putreal(val)
      subroutine getstr(string,l)
      subroutine getint(ival)
      subroutine getreal(val)
```

Fig. 12 Procédures d'interface PROLOG-FORTRAN développées pour PIGAL :
en haut : un exemple de l'appel de la subroutine "write"
au milieu : la règle 1 transforme le nom de la subroutine dans l'argument du prédicat externe CALLSUB, les règles 2-8 sont les 7 prédicats externes assurant cette interface
en bas : la subroutine user_rule, l'appel 'call callsub' se trouve au label 560

3.3 Fonctionnalités de PIGAL et leurs réalisations

Les fonctionnalités sont sous le contrôle direct du programme principal en PROLOG, qui assure le logique de l'ensemble du système. Le rôle du programme PROLOG consiste en premier lieu à vérifier la validité de la commande : compatibilité avec la grammaire du langage de commande, et compatibilité des éléments composant une sélection. A chaque niveau de l'analyse les erreurs sont signalés et les corrections possibles sont proposées. La commande est alors convertie en une liste d'appels de subroutines, avec les arguments adéquats, pour effectuer l'action demandée.

Les fonctions les plus significatives sont les suivantes :

- représentation d'événement en graphique 3D
- sélection multiple d'entités selon des coupures physiques et recherche de relations indirectes entre entités
- reconstruction interactive
- manipulation interactive d'histogrammes
- sélection d'événements et création de répertoire d'événements

Chaque fonctionnalité est réalisée soit en pur PROLOG, soit par combinaison de PROLOG avec FORTRAN.

Dans les sections suivantes ces fonctionnalités et leurs réalisations seront présentées. La présentation utilise certaines notions de graphique dont l'explication se trouve dans l'Annexe 4.

Représentation d'événement en graphique 3D

PIGAL est un logiciel qui peut représenter l'image d'événements d'ALEPH en graphique 3D. La Fig.13 est une image d'événement issue des commandes suivantes:

```
draw/col=cyan saea          (Contour du détecteur à petit angle)
draw/col=cyan saeb          (                -                  )
```

```
draw/col=cyan lcea          (Contour des moniteurs de luminosité)
draw/col=cyan lceb          (                -                  )
draw/col=cyan tpae                  (Silouhette TPC)
draw/col=cyan tpai                  (     -       )
draw/col=cyan tpbe                  (     -       )
draw/col=cyan tpbi                  (     -       )
draw/col=cyan ecbl                  (Contour ECAL )
draw/col=cyan hcbl                  (Contour HCAL)
draw/col=red pfrf                   (Trace chargée)
draw/col=blue peco without pfrf
                                (Objet ECAL non associé à une trace)
draw/col=red peco with pfrf
                                (Objet ECAL associé à au moins une trace)
draw/col=blue phco                  (Objet HCAL)
draw/col=green tpco                 (Coordonnée du damier de TPC)
```

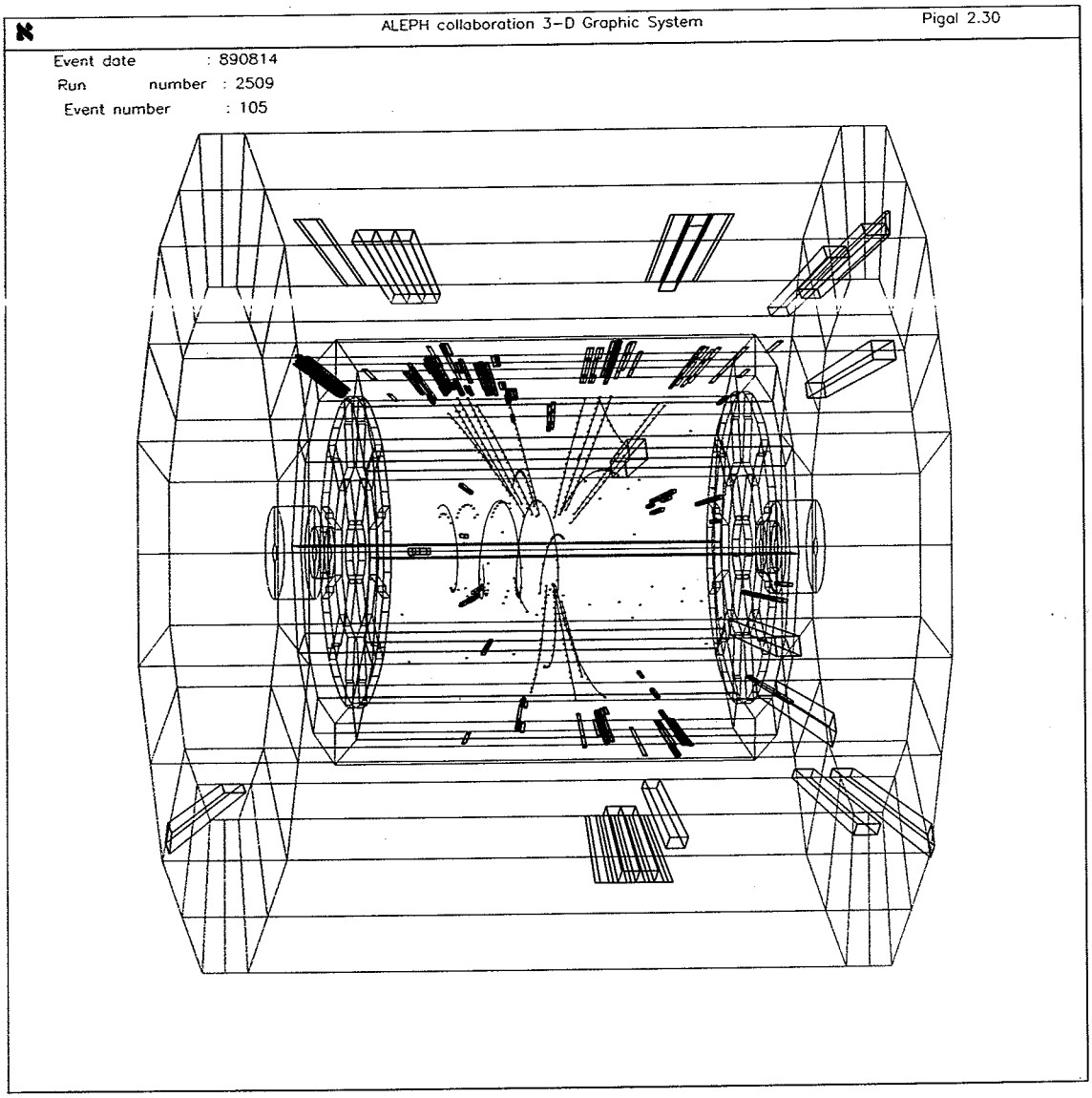


Fig. 13 L'image d'un événement d'ALEPH, représentée par PIGAL

PIGAL permet de

- représenter un événement complet ou partiel (sous-détecteur spécifié, un ensemble d'entités sélectionnées selon les coupures physiques)
- sélectionner la représentation ou projection (X Y Z, θ Φ, R Z)
- sélectionner les caractéristiques graphiques (couleur, fenêtre ...)

Actuellement, plus de 60 entités peuvent être dessinées, des contours du détecteur aux événements après la reconstruction.

Une fois l'image représentée l'utilisateur peut interagir directement avec l'image, pointer avec la souris de segments sur l'écran (voir l'Annexe 4 pour la notion de "pick") pour obtenir les informations internes liées aux objets graphiques (par ex. les données liées à une trace), et effectuer l'analyse de l'événement de façon plus générale. Par exemple

commande: `draw/col=red track`

opérations: représenter les traces en couleur rouge

commande: `print track pick`

opérations:
- demande une opération de pointage sur l'entité track qui est déjà visualisée
- imprime les données internes d'une trace choisie par l'utilisateur

Réalisation

L'ensemble des fonctions graphiques est réalisé par un module graphique écrit en FORTRAN, et qui constitue la moitié du code PIGAL (la Fig.14). Cette partie est complétement fermée, une fois obtenues les informations minimum nécessaires (entité, sélecteur) la production d'image se fait de façon automatique.

Pour que PIGAL s'adapte à différentes stations graphiques 3D et à différentes bibliothèques graphiques commerciales, une interface de programme d'application indépendante des bibliothèques graphiques a été développée. La position de cette interface est indiquée dans la Fig.14. Elle définit une série de **macroprimitives** et un ensemble de procédures pour les manipuler. La notion de "macroprimitive" vient de

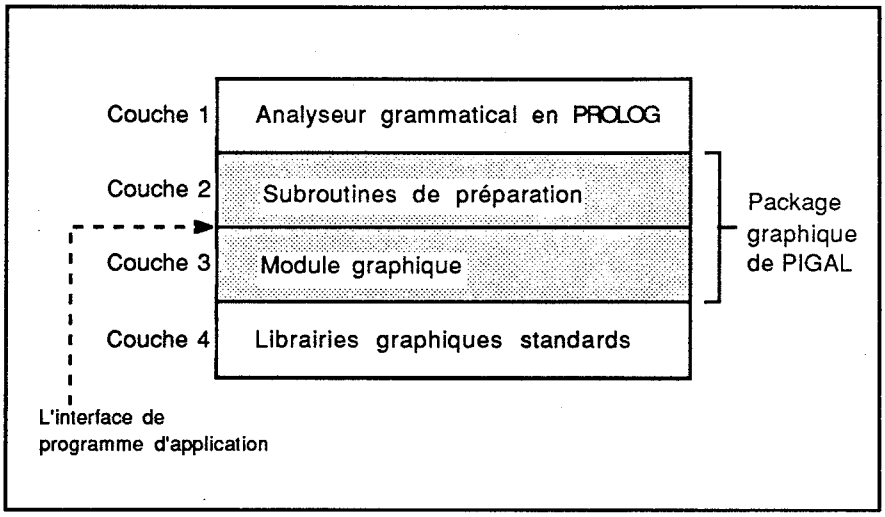


Fig. 14 La position du module graphique dans PIGAL : les couches 2 et 3 constituent ce package, entre elles il y a une interface séparant le programme d'application des bibliothèques graphiques.

la notion de "primitive" dans GKS. Les primitives sont les éléments graphiques acceptés par GKS : polyline, texte, surface, polymarker ... Pour PIGAL, on a ajouté dans cette base certains éléments graphiques utilisés souvent dans l'analyse de données de la physique expérimentale, par exemple : hélice. Ces deux parties des éléments graphiques plus le mélange de ces primitives constituent les macroprimitives de PIGAL.

Le système est divisé en 4 couches (Fig.14). L'analyseur grammatical en PROLOG (couche 1) digère la commande de l'utilisateur, fait appel à une subroutine de la couche 2. La subroutine transforme les données (stockées dans une entité) en macroprimitives. Celles-ci seront transformées ensuite, par les procédures de la couche 3, en primitives graphiques.

Précisons le fonctionnement de ces 4 couches, en analysant la commande :

```
draw tpco
```

(représenter les coordonnées du damier de TPC)

Couche 1 : Analyse de commande

Une partie de la base de faits PROLOG concerne la relation entre verbe, entité et nom de la subroutine effectuant sa représentation, qui a la forme:

```
prepare-routine(entity-name,subroutine-name,verb)
```

La Fig.15 décrit une partie de ces règles. Pour la commande

```
draw tpco
```

on vérifie d'abord que **tpco** soit une entité représentable, puis que la relation entre **draw** et **tpco** soit présente. Ces conditions étant remplies, on exécute la subroutine RTPCO située dans la couche 2.

```
prepare-routine(tpco,rtpco,draw,nil) ->;
prepare-routine(etp1,retp1,draw,nil) ->;
prepare-routine(tgco,rtgco,draw,nil) ->;
prepare-routine(kin2,rkin2,draw,nil) ->;
prepare-routine(ver2,rver2,draw,nil) ->;
prepare-routine(fkin,rfkin,draw,nil) ->;
prepare-routine(fver,rfver,draw,nil) ->;
prepare-routine(tgft,rtgft,draw,nil) ->;
prepare-routine(frft,rfrft,draw,nil) ->;
prepare-routine(x,y,highli,z) -> prepare-routine(x,y,draw,z);
prepare-routine(x,rerase,erase,nil) -> prepare-routine(x,y,draw,z);
```

Fig. 15 La relation entre verbe, entité et nom de subroutine effectuant le travail, par exemple : la première règle indique que PROLOG fait appel la subroutine RTPCO pour dessiner l'entité TPCO

Couche 2 : Subroutines de préparation

La couche 2 contient 64 subroutines de préparation, chacune correspond à une entité représentable (ex. **tpco**). Les données de l'entité, déjà choisies par le sélecteur **dsel** (page 49), seront transformées en macroprimitives par l'une de ces subroutines. La Fig.16 présente une partie de la

subroutine RTPCO qui transforme les données de l'entité tpco en macroprimitive "polymarker" par l'appel de la procédure DSMARK.

```
      DATA BNAME/'TPCO'/

Get the selected bank for the entity TPCO
      CALL RDSEL(NDSEL,BNAME)
      IF(NDSEL.EQ.0) RETURN
      KDSEL = NLINK('DSEL',NDSEL) + 1
      NLSEL = LROWS(KDSEL)

Access number and family number of the selected entities
      DO 70 IL = 1, NLSEL
        NLINE = ITABL(KDSEL,IL,1)
        IR = ITABL(KDSEL,IL,2)

Check that this entity is present in the data structure
        KTPCO = NLINK(BNAME,IR)
        IF(KTPCO.EQ.0) THEN
          WRITE(6,*) 'NO ENTITY',BNAME,' AVAILABLE'
          GOTO 999
        ENDIF

Get space coordinates
        NBPT = 1
        RO = RTABL(KTPCO,NLINE,JTPCRV)
        PHI = RTABL(KTPCO,NLINE,JTPCPH)
        XYZ(1) = RO*COS(PHI)
        XYZ(2) = RO*SIN(PHI)
        XYZ(3) = RTABL(KTPCO,NLINE,JTPCZV)

Request an entity in graphics structure to store points
        NBPRIM = 1
        CALL DSMARK(BNAME,NLINE,IR,NBPRIM,NBPT,XYZ)

Check polymarker changes in the current command
        CALL GETQUA('MAR',ITYP,IDUM,POLMAR,TDUM)
        CALL GETQUA('MSI',ITYP,IDUM,SIZMAR,TDUM)
        CALL GETQUA('COL',ITYP,IDUM,COLIND,TDUM)

Report polymarkers changes
        IF(POLMAR.NE.0)
     +  CALL GMODM(BNAME,NLINE,IR,NBPRIM,'TYPE',POLMAR)
        IF(SIZMAR.NE.0)
     +  CALL GMODM(BNAME,NLINE,IR,NBPRIM,'SIZE',SIZMAR)
        IF(COLIND.NE.0)
     +  CALL GMODM(BNAME,NLINE,IR,NBPRIM,'COLO',COLIND)

70    CONTINUE
```

Fig. 16 Une partie de la subroutine RTPCO qui produit le macroprimitive 'marqueur' en appelant la procédure DSMARK, et change les attributs graphiques en utilisant la procédure GMODM.

Interface entre Couche 2 et Couche 3

DSMARK est l'une de procédures de l'interface entre la couche 2 et la couche 3, qui sépare le programme d'application d'avec les bibliothèques graphiques concrètes. Grâce à cette interface, PIGAL peut être adapté aux derniers développements de logiciels graphiques sans que le code d'application ait besoin d'être changé.

L'interface contient autant de procédures que de macroprimitives possibles :

DSLINE	(pour Ligne)
DSSURF	(pour Surface)
DSMARK	(pour Marqueur)
DSTEXT	(pour Texte)
DSHIST	(pour Axe d'histogramme)
DSHEL	(pour Hélice)

• • •

et une série de procédures permettant de changer les attributs graphiques :

GMODL	(changer les attributs graphiques de Ligne)
GMODM	(pour Marqueur)
GMODS	(pour Surface)
GMODT	(pour Axe d'histogramme)
GMODH	(pour Hélice)

• • •

Couche 3 : Module graphique de PIGAL

Le module graphique est la dernière couche avant d'accéder à la bibliothèque graphique. Il gère un ensemble de structures de données graphiques concernant les objets représentables, ainsi que leurs relations avec l'entité originale (ex. tpco). Ces relations permettent de manipuler l'image graphique.

Quand l'on pointe un segment sur l'écran avec la souris, l'information saisie peut remonter directement jusqu'à l'analyseur grammatical. Celui-ci traduit l'information en une sélection formelle, qui fait référence à

l'objet pointé. Lorsque un objet complexe est concerné, cette traduction fait intervenir la relation entre objet complexe et objet représentable, décrite dans la section de la base de faits qui représente la structure graphique hiérachique (page 32).

Couche 4 : bibliothèque graphique

Grâce aux couches 2 et 3 qui séparent le code d'application d'avec les bibliothèques graphiques, PIGAL a une grande facilité d'adaptation aux différents produits de logiciels graphiques :

- UIS (User Interface Software) de DEC : il fournit un ensemble de fonctions permettant de piloter les graphiques et les opérations de fenêtrage.

- GKS, le standard graphique.

- SGR (Structured Graphics Routines) : il permet la création de structures de données graphiques en mémoire dynamique. Ce système cohabite avec le système X-WINDOWS qui assure la gestion des fenêtres et des tables de couleurs de la station VAX. Les deux bibliothèques coopèrent pour offrir des services graphiques de haute qualité et notamment un graphisme en trois dimensions très performant.

Sélection d'entités sur critères multiples et recherche de relation indirecte entre entités

Les objets à représenter peuvent être sélectionnés selon :

- **un attribut,** par exemple pour sélectionner les traces ayant une impulsion supérieure à 10GeV/C, on tape la commande :

```
tracks momentum>10
```

- **une relation,** par exemple pour sélectionner les cellules du calorimètre électromagnétique qui sont dans la même cluster, on tape la commande :

```
esda in eclu
```

- **critères multiples,** par exemple pour sélectionner les traces ayant une

impulsion supérieure à 10GeV/C et n'étant associées avec aucune cellule du calorimètre électromagnétique (Ecal storey data), on tape la commande :

```
tracks momentum>10 without calobject
```

et pour sélectionner les traces ayant une impulsion supérieure à 10GeV/C et portant une charge positive, on tape la commande :

```
tracks momentum>10 and charge>0
```

- **relation indirecte**, par exemple pour sélectionner les clusters chargés, on tape la commande

  ```
  cluster with track
  ```

  puisque **cluster** et **track** n'ont pas de relation directe, le programme propose un chemin possible. Lorsque l'utilisateur a accepté le chemin proposé, la sélection indirecte est remplacée, pour l'exécution, par une sélection explicite, mettant en jeu le chemin et l'entité concernée. Dans cet exemple, c'est l'entité **erl3** qui fait la liaison entre ces deux entités originales, donc la commande devient

  ```
  cluster with erl3 in track
  ```

- **attribut indirect**, par exemple pour sélectionner sur un attribut qui n'est pas l'attribut de l'entité choisie,

  ```
  cluster momentum>4
  ```

  où **momentum** n'est pas un attribut de **cluster**, le programme cherche les entités ayant une relation avec **cluster** et possédant un attribut **momentum**. Avec le même mécanisme que dans l'exemple précédent la commande devient

  ```
  cluster with erl3 in track momentum>4
  ```

Relations entre entités

Les données d' ALEPH sont organisées dans la cadre du modèle Entité-Association, comme décrit dans le chapitre 2. La relation entre entités est soit **un-à-plusieurs**, soit **plusieurs-à-un.** Dans l'exemple de la Fig.7, la double flèche signifie une relation **plusieurs-à-un.** Le support de la relation est toujours situé du coté **plusieurs**, par exemple la relation **esda-eclu** est représentée dans l'entité **esda** par la neuvième colonne (voir la Fig.10, page 33). Si la relation réelle entre deux entités est

plusieurs-à-plusieurs, qui n'est pas permise par ADAMO, une entité intermédiaire est créée qui porte une relation **plusieurs-à-un** à chaque entité originale [16]. Par exemple un cluster de ECAL peut être touché par plusieurs traces, et une trace peut traverser plusieurs clusters, donc leur relation est plusieurs-à-plusieurs, ce qui ne correspond pas au modèle d'ADAMO. On utilise l'entité **erl3** pour transformer cette relation en deux relations plusieurs-à-un.

PIGAL utilise deux termes **in** et **with** pour décrire la relation **plusieurs-à-un** selon qu'elle suit la flèche dans l'un ou l'autre sens, par exemple un cluster comporte plusieurs cellules, on peut donc dire :

```
esda in eclu     ou    eclu with esda
```

Les sélections complexes peuvent être effectuées à partir de 4 familles d'opérations élémentaires **in, out, with, without**. Grâce à ce mécanisme, on peut utiliser des commandes comportant une série de sélections. Par exemple pour l'opération : " imprimer toute trace associée à un bloc de cellules du calorimètre, dont l'une au moins a une énergie supérieure à 500 Mev et dont l'angle théta est plus grand que 0.3 radians", on tape la commande

```
Print tracks with clusters with cells energy>0.5 and
theta>.3
```

Niveau PROLOG

Après l'analyse de la commande, le programme PROLOG appelle dans l'ordre adéquat les subroutines de la bibliothèque SELTAB en utilisant les bons arguments.

Dans le cas d'une relation indirecte, le programme doit exploiter le dictionnaire DDL pour chercher le chemin permettant d'expliciter la relation entre deux entités. Ce mécanisme de "navigation" fait appel à deux éléments fondamentaux du langage PROLOG : unification et récursivité (voir l'Annexe 3). L'idée principale se trouve dans la Fig.17.

```
règle 1        chemin(a,b,c,l) -> relation(a,b,c);
règle 2        chemin(a,b,c1.x.c2,l) -> relation(a,x,c1)
                                        hors(x,l)
                                        chemin(x,b,c2,x.l);

règle 3        relation(a,b,in) -> attribut(a,b,3,x);
règle 4        relation(a,b,with) -> attribut(b,a,3,x);

règle 5        hors(x,nil) ->;
règle 6        hors(x,x.l) -> impasse;
règle 7        hors(x,y.l) -> dif(x,y)hors(x,l);
```

Fig. 17 Un exemple en PROLOG pour la recherche des relations indirectes entre entités. Ici **a**, **b**, **x** sont les variables instanciées par le nom d'entité.
Règle 1 : "chemin(a,b,c,l)" est réussi s'il existe une relation entre **a** et **b**
Règle 2 : "chemin(a,b,c1.x.c2,l) est réussi s'il existe une relation entre **a** et **x** et si "chemin(x,b,c2,x.l)" est réussi. C'est une règle récursive qui va verifier s'il existe une relation entre **x** et **b**, ainsi de suite. Dans cette règle, **l** est une liste contenant les entités déjà examinées, le terme "hors(x,l) verifie que **x** ne fait pas partie de la liste **l**, ce qui evite une recherche en boucle sans fin.

Recherche de relation au niveau FORTRAN - Bibliothèque SELTAB

L'ensemble des subroutines de SELTAB travaille sur une table de deux colonnes dite **sélecteur**. La Fig.18 donne un exemple de sélection. Le sélecteur porte en première colonne les numéros de ligne de l'entité à sélectionner, en deuxième colonne les numéros de ligne de l'entité référenciée. Dans cet exemple, **esda** est l'entité qu'on sélectionne et **eclu** est l'entité référenciée. En tête de chaque colonne figure le nom de l'entité correspondante.

Une fois le sélecteur formé, sous le nom de **dsel**, il devient la base de toutes les opérations suivantes : draw, print, histogram, erase, etc.

Ce sélecteur est utilisé à la fois en entrée et en sortie par chacune des subroutines de SELTAB. Ceci permet d'effectuer des sélections successives sur la même entité.

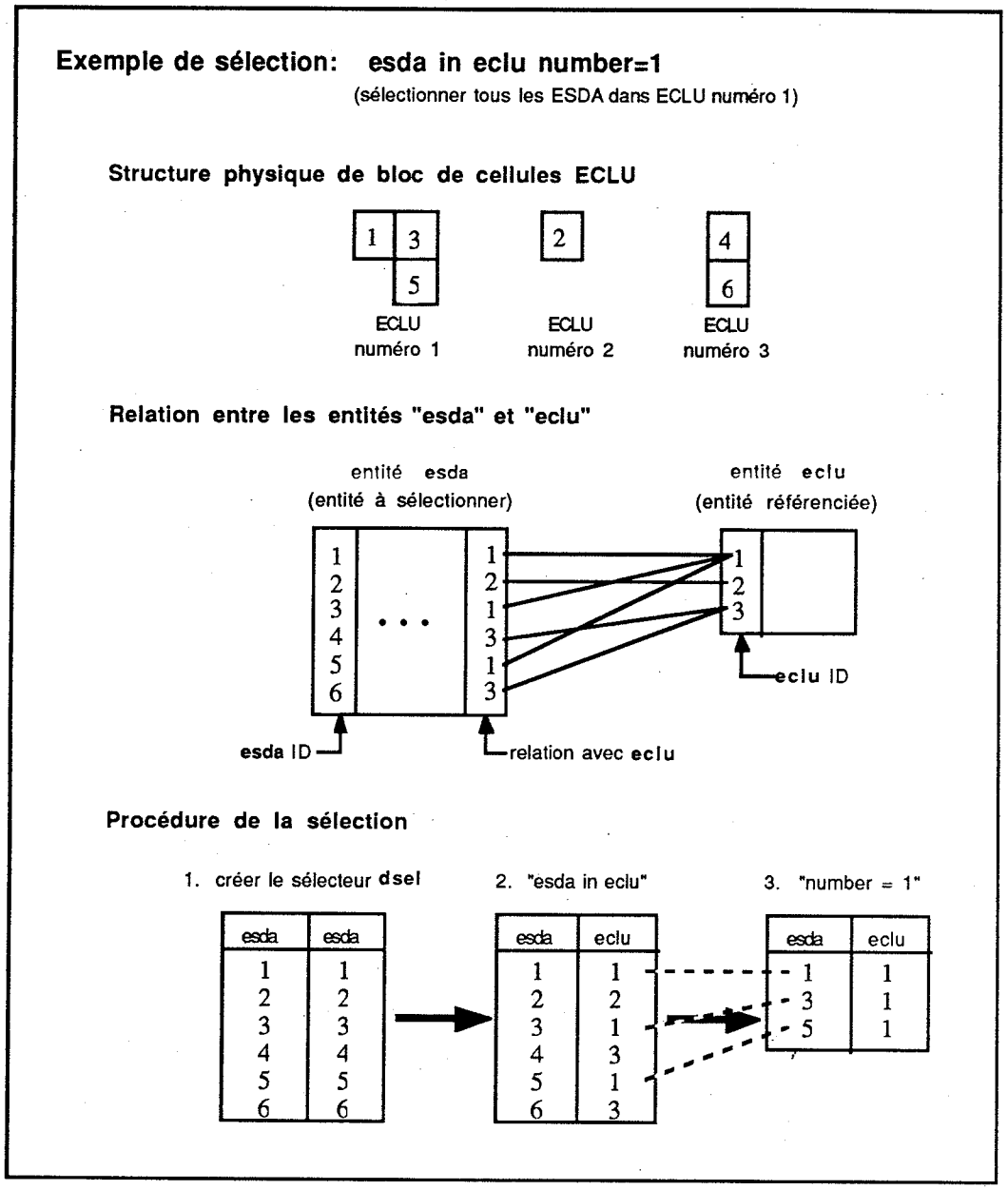


Fig. 18 Le mécanisme de la sélection

partie 1 : Structure physique de bloc de cellules "ECLU" - chaque "ECLU" contient un certain nombre de cellules "ESDA"

partie 2 : Organisation des données - les données de ESDA et de ECLU sont classées dans deux entités **esda** et **eclu**. L'entité **esda** contient la relation avec l'entité **eclu**, par ex. "esda" 3 est dans "eclu" 1, "esda" 4 est dans "eclu" 3 ...

partie 3 : Procédure de la sélection - fabrication du sélecteur

1. créer un sélecteur nommé "**dsel**" avec deux colonnes, les remplir par l'identificateur de l'entité **esda**
2. quand on effectue "esda in eclu", la deuxième colonne est remplacée par la colonne de relation située dans l'entité **esda**
3. effectuer "number=1", sélectionner sur la colonne "eclu" les lignes dont le contenu est 1

Le sélecteur **dsel** contient bien les cellules 1, 3, 5, qui correspondent au premier agrégat.

Les principales subroutines de sélection sont :

- SELTAB, qui effectue les conditions >, <, = , par exemple

```
momentum>5
```

- SELIN, qui gère la relation **in**, par exemple

```
esda in eclu
```

- SELWITH, qui gère la relation **with**, c'est le sens inverse de **in**, par exemple

```
track with vertex
```

Quelques subroutines supplémentaires permettent d'initialiser le sélecteur, de manipuler une pile de sélection pour résoudre les problèmes liés au parenthésage dans les expressions. Les sélections exclusives: **without**, **out** sont aussi possibles.

Reconstruction interactive

PIGAL est conçu pour s'adapter à un ficher du type POT (DST) qui est la sortie de JULIA, aussi bien qu'aux données brutes. Dans le second cas, le système peut accepter des commandes faisant référence à certaines entités élementaires inexistantes (par exemple des traces) et les reconstruire si besoins avant l'exécution des commandes. Ceci s'appele "reconstruction interactive".

Réalisation

Lors de la lecture d'un événement, un examen des diverses entités présentes est effectué. Les noms des entités présentes sont mémorisés au moyen de la règle de PROLOG **`assert`**, ex.

```
entite-possible(x)
fortran(depres,x.1.nil,y.z.nil)
eq(z,1)
assert(entite-presente(x),nil);
```

ici la subroutine DEPRES vérifie si l'entité **x** existe. Si z=1 (**x** existe), le

programme ajoute une règle

```
entite-presente(x)
```

dans la base de connaissances qu'il pourra utiliser ultérieurement. Pendant l'exécution, lorsqu'une action doit être effectuée sur une entité non-existante, cette entité peut être reconstruite dynamiquement en appelant les subroutines nécessaires de JULIA. Par exemple l'entité tpco (coordonnées du damier de TPC) ne se trouve pas directement dans le fichier de données, il faut la reconstruire avant de dessiner.

Les connaissances sur le programme de reconstruction sont regroupées sous forme de règle de différents niveaux (la Fig.19) et de la forme :

```
module(entités d'entrée,entités de sortie,nom de routine)
```

qui contient trois types d'information :

- une liste des entités d'entrées de cette procédure
- une liste des entités créées par la procédure
- le nom d'une procédure de JULIA

```
event-module(esda.etp1.nil,erl3.eclu.nil,ecfclu) ->;
event-module(hsda.nil,hclu.nil,hcfclu) ->;
event-module(eclu.nil,ebos.ecob.nil,ecfobj) ->;
event-module(eclu.erl3.nil,eidt.nil,elecid) ->;
event-module(ecob.hclu.etrx.nil,crl3.calo.nil,echcgl) ->;
event-module(nil,esda.nil,epreda) ->;
event-module(nil,hsda.nil,hpreda) ->;
event-module(nil,phst.nil,hpreda) ->;

run-module(nil,nil,einiru) ->;
run-module(nil,nil,hiniru) ->;
run-module(nil,eslo.nil,deeslo) ->;
run-module(nil,hslo.nil,crhslo) ->;

job-module(nil,nil,einijo) ->;
job-module(nil,nil,hinijo) ->;
job-module(nil,nil,cinijo) ->;
```

Fig. 19 Les connaissances sur le programme de reconstruction JULIA sont groupées sous forme de module de 3 niveaux : event (exécuter une fois par chaque événement), run (exécuter une fois par chaque run), job (exécuter une fois par chaque job).

Supposons que la base de faits sur JULIA soit :

```
règle 1:    module(obj1.obj2.obj3.nil,obj4.obj5.nil,sub1)
règle 2:    module(          obj7.nil,     obj3.nil,sub2)
régle 3:    module(               nil,obj1.obj2.nil,sub3)
règle 4:    module(               nil,     obj7.nil,sub4)
```

Le mécanisme de recherche d'une entité est presenté dans l'exemple suivant, pour une commande

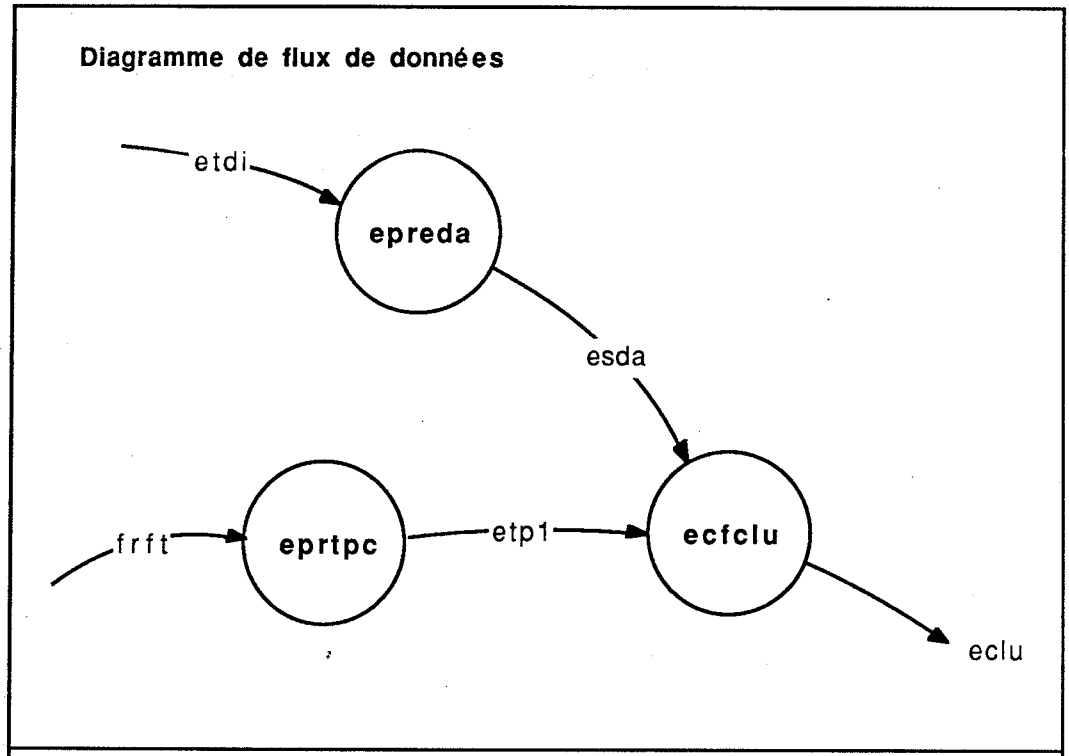
```
draw obj5
```

PIGAL effectue une recherche de obj5 par les étapes suivantes :

Action	**Résultat**
Vérifier les entités existantes	**obj5** absente
Chercher **obj5** dans les listes d'entité produites	Trouve **obj5** dans la règle 1
Verifier la liste des entités d'entrée du module	Trouve **obj1,obj2,obj3** [qui doivent exister avant **obj5**]
Chercher **obj1** dans les listes d'entités produites	Trouve **obj1** dans la règle 3, la liste des entités d'entrée du module est vide (règle 3). Donc **obj1** est évaluable par appel à sub3.
Appeler sub3	produire **obj1,obj2**
Chercher **obj3**	Trouve la règle 2
Chercher **obj7**	Trouve la règle 4, la liste des entités d'entrée du module est vide (règle 4)
Appeler sub4	produire **obj7**

Appeler sub2	Produire **obj3**
Appeler sub1	produire **obj5**

On trouvera Fig.20 une partie de DFD (Diagramme de flux de données) concernant la reconstruction de l'entité **eclu** et les règles PROLOG correspondantes.



Règles PROLOG

```
règle 1    module(epreda, etdi.nil, esda.nil) ->;
règle 2    module(ecfclu, esda.etp1.nil, eclu.nil) ->;
règle 3    module(eprtpc, frft.nil, etp1.nil) ->;
```

Fig. 20 L'exemple de la reconstruction interactive
en haut : une partie de DFD présentant le processus de la reconstruction de l'entité ECLU
en bas : les trois règles de PROLOG assurant ce travail. ECLU est l'entité de sortie de la règle 2 qui a besoin des entités ESDA et ETP1 comme arguments d'entrée. ESDA et ETP1 sont les entités de sortie de la règles 1 et 3 respectivement. Donc les séquences d'exécutions décidées par PROLOG sont : appel à EPREDA, puis appel à EPRTPC, et en fin appel à ECFCLU.
Le traitement est contrôlé par les opérations : une opération n'est effectuée que lorsque son résultat est nécessaire à une autre opération.

Manipulation interactive d'histogramme

Depuis très longtemps les physiciens du CERN manipulent les histogrammes de façon non interactive, au moyen du logiciel HBOOK [31]:

```
call hbook1(id,chtitl,nx,xmi,xma,vmx)
call hbook2(id,chtitl,nx,xmi,xma,ny,ymi,yma,vmx)
call hfill(id,x,y,weight)
```

Depuis deux ans on utilise PAW [24], un système interactif qui gère les opérations de HBOOK et permet la représentation d'histogramme sur différentes stations graphiques. L'utilisateur peut manipuler des histogrammes par menu ou par commande:

```
paw> h/create/1dhisto id title ncx xmin xmax
paw> h/create/2dhisto id title ncx xmin xmax ncy ymin
     ymax
paw> v/hfill vname id
```

Au niveau de la création et du remplissage il n'a pas de changement essentiel par rapport à HBOOK, c'est à dire que le remplissage n'est pas automatique, l'utilisateur doit préparer les données dans un vecteur avant l'appel aux procédures de HBOOK. De plus, PAW possède sa propre structure de données (N-tuples) qui ne possède pas la richesse des systèmes comme ADAMO ou BOS, adaptés à la manipulation des événements de physique des particules. Avant d'utiliser PAW il est donc nécessaire de préparer les données.

PIGAL essaye d'améliorer ces deux points, il permet de

- créer les histogrammes interactivement en utilisant les noms mnémoniques des quantités physiques
- remplir les histogrammes automatiquement.

La syntaxe de commande d'histogramme est :

```
histogram[/qualifiers] attribute [versus attribute] of
object
```

La création et le remplissage des histogrammes sont simultanés. Quelque

exemples sont listés ici :

- Histogramme pour un seul événement : pour créer et remplir un histogramme représentant l'énergie des cellules du calorimètre électromagnétique, on utilise

  `histogram energy of esda`

  pour créer et remplir un histogramme représentant l'angle théta des cellules du calorimètre électromagnétique affecté d'un poids **energy**, on utilise

  `histogram/wei=energy theta of esda`

  pour créer et remplir un histogramme à deux dimensions, on utilise

  `histogram energy versus phi of esda`

- Histogramme sur plusieurs événements

```
macro mev                          (déclaration de la macro mev)
   loop                            ( boucler sur tous les événements
                                   d'un fichier de données d'ALEPH)
     histogram/id=100 number of track
                                   (remplir un histogramme du
                                   nombre de traces de chaque
                                   événement)
     xgetev                        (lire le prochain événement)
   endloop                         (fin de la boucle)
endmacro                           (fin de la macro mev)
```

L'image de l'histogramme (effectuée sur un seul événement) permet ensuite de manipuler l'événement par pointage avec la souris, par exemple (Fig.21) : supposons que l'utilisateur ait déjà créé un histogramme ayant pour identificateur ID=10

commande: `draw/win=2 hist id=10`

opération : dessiner sur la deuxième fenêtre cet histogramme

commande: `draw/win=1/col=red track in range pick`

opération :
- demande deux opérations de pick sur l'histogramme pour sélectionner une zone
- représente en couleur rouge les traces **track** qui sont représentées dans la zone indiquée.

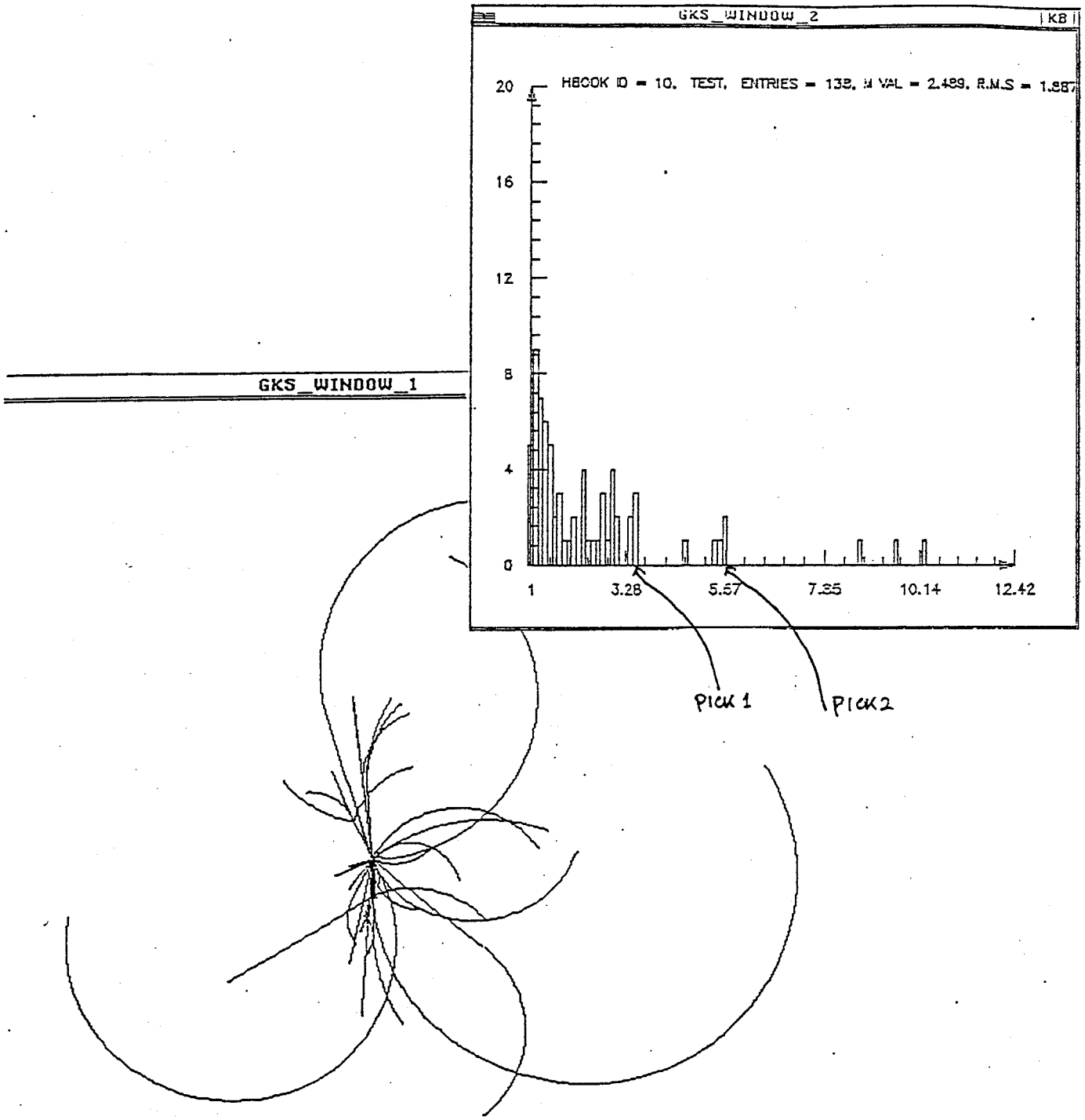


Fig. 21 Manipulation de la représentation de l'événement par l'histogramme.

C'est un écran issu des commandes suivantes :
draw track momentum>1 (représenter les traces avec la condition imposée)
histogram/id=10/cht='test/nx=100 momentum of track
(créer et remplir l'histogramme par impulsion de traces)
draw/win=2 hist id=10 (représenter l'histogramme dans la fenêtre 2)
draw/win=1/col=red track in range pick
(On demande une représentation de trace avec les conditions imposées par "in range pick". Ce qui implique deux opérations de pick sur l'histogramme dessiné dans la fenêtre 2 pour sélectionner le minimum et le maximum des impulsions. Ensuite PIGAL représente en couleur rouge les traces dont l'impulsion est représentée dans cette zone.)

Réalisation

Dans PIGAL, les histogrammes sont incorporés dans le modèle ER, les informations sont organisées en trois entités internes **hist**, **dbin** et **pick** (voir la section 1 de la Fig.22) :

- l'entité **hist** contient les informations concernant la structure de tous les histogrammes existants dans HBOOK à un moment donné (identificateur, titre de l'histogramme, poids, etc.)
- chaque ligne de l'entité **dbin** contient la valeur d'un canal de l'histogramme choisi.
- L'entité **pick** permet de mettre en relation l'histogramme et l'objet qu'il représente (par ex. la trace).

Un module d'histogramme manipule ces trois entités et les met en relation avec HBOOK. Les entités **hist** et **dbin** sont ensuite transmises comme des entités élémentaires et peuvent être dessinées par le programme graphique.

Les créations et les emplois de ces trois entités internes sont décrits ci-dessous.

Création et remplissage d'un histogramme

Après l'analyse de la commande, le programme PROLOG fait appel au module d'histogramme qui crée et remplit l'histogramme. Ce module reçoit le nom de l'entité, les noms des attributs, leurs types, ainsi que l'information de poids éventuelle, par exemple, la commande

```
histogram/weight=energy theta of esda
```

crée et remplit un histogramme représentant l'angle **theta** des cellules du calorimètre électromagnétique affecté d'un poids **energy**. **Energy** est un attribut de l'entité **esda**.

Avec cet ensemble d'informations, le module d'histogramme fait une boucle sur les lignes de l'entité choisies par le sélecteur **dsel** (page 49), avec ou sans poids, remplit l'histogramme en appelant les procédures de HBOOK.

Une fois l'histogramme construit, toutes les informations numériques sont stockées dans les variables internes de HBOOK. Le module d'histogramme ne garde, dans l'entité **pick**, qu'un descripteur interne contenant les informations concernant l'identificateur de l'histogramme, le nom de l'entité histogrammée et les attributs choisis. Par exemple pour la commande

```
histogram/id=100 energy of esda
```

les informations gardées dans le module d'histogramme sont l'identificateur **100**, le nom de l'entité **esda** et l'attribut **energy** avec son type.

Représentation d'un histogramme

Le module d'histogramme récupère les informations numériques dans HBOOK au moment où l'utilisateur le demande, par exemple

```
draw hist id=100
```

Le module d'histogramme travaille étape par étape :

- créer les entités **hist** et **dbin** (s'il s'agit de la première commande concernant l'histogramme)
- obtenir les informations à l'aide de procédures de HBOOK
- transmettre ces informations dans les entités **hist** et **dbin**

Ces deux entités sont ensuite dessinées par le programme graphique.

Relation entre la représentation de l'événement et la représentation de l'histogramme

Dans le dessin d'histogramme, chaque canal est représenté par un polyline mis en relation avec un segment graphique. Quand l'on pointe un canal (voir la section 2 de la Fig.22), le programme utilise l'entité **dbin** pour trouver l'identificateur de l'histogramme et la valeur du canal, et l'entité **pick** pour trouver le nom d'entité (par ex. "track" en Fig.21) et les noms des attributs (par ex. "momentum"). Les informations trouvées sont transmises au programme principal. Celui-ci forme une

nouvelle commande avec une sélection précise.

C'est ainsi que la commande typique :

```
draw track in range pick
```

peut être traduite en

```
draw track a>momentum and momentum<b
```

où **a** et **b** sont les valeurs minimales et maximales déterminées par pointage avec la souris. Cette commande sera à son tour soumise à l'analyse et exécutée (voir la Fig.21, page 57).

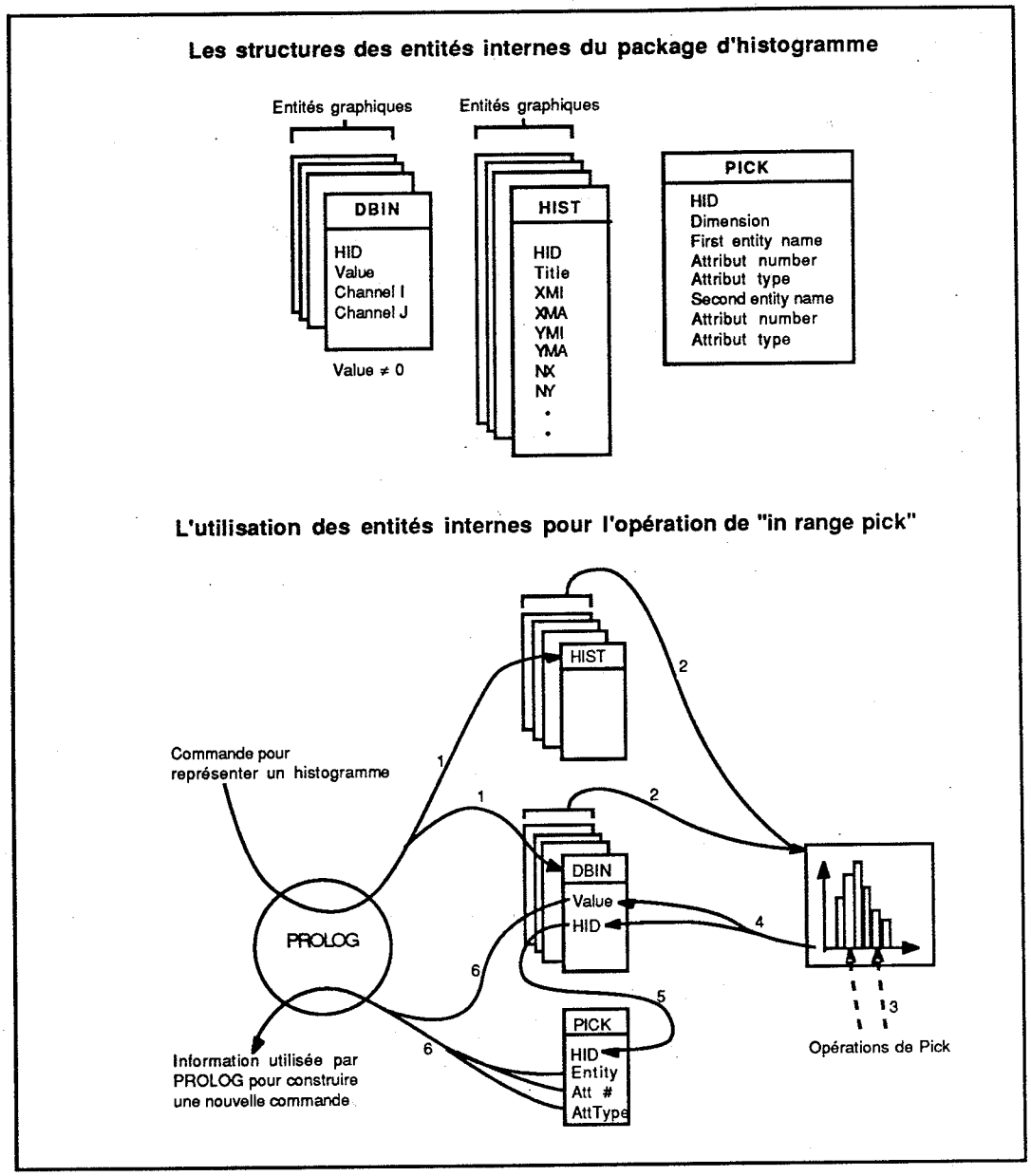


Fig. 22 Les structures de données internes du module d'histogramme.
en haut : Structure de trois principales entités internes utilisées par le module d'histogramme de PIGAL. Derrière les entités dessinables (HIST, DBIN) sont figurées les entités graphiques utilisées par le module graphique de PIGAL (page 45)
en bas : Séquence des opérations de la fonction "in range pick"; les ordres sont :

1. créer et remplir HIST et DBIN, créer et remplir les entités graphiques correspondantes
2. représenter l'histogramme en utilisant les informations stockées dans les entités graphiques
3. l'opérateur effectue deux "pick" sur l'image de l'histogramme pour sélectionner une zone
4. trouver dans l'entité DBIN l'identificateur de cet histogramme, et les valeurs des bins
5. trouver dans l'entité PICK le nom de l'entité (**track** pour l'exemple de Fig.21) ainsi que l'attribut original (**momentum**) de cet histogramme
6. le nom de l'entité, le numéro de l'attribut, le type de l'attribut et les valeurs de deux bins retournent à PROLOG pour former une nouvelle commande

Sélection d'événement et création de répertoire d'événement

La sélection peut s'effectuer avec ou sans condition physique.

- **Sans condition :**
  Pour lire dans le fichier "zzero.edir" un événement ayant un numéro de run et un numéro d'événement donnés, on emploie la commande :

```
xgetev/evt=300/run=4589/fil='zzero.edir'
```

- **Avec condition :**
  L'utilisateur peut classer les événements selon un critère physique de son choix et créer son propre répertoire d'événements. L'exemple suivant permet de chercher dans un fichier de données tous les événements contenant au moins un électron :

```
macro electron
  sevt/dir='electron.edir'
  loop
    xgetev
    sevt track momentum>10 with eidt ippoth=1
  endloop
endmacro
```

ici **electron** est le nom de la macro, **eidt** est l'entité d'électrons, **ippoth=1** signifie un électron. Donc la condition de sélection est : "chercher les événements qui contiennent au moins un électron dont l'impulsion est supérieure à 10GeV/C ". Les événements sélectionnés sont enregistrés dans un fichier 'electron.edir' et peuvent être utilisés ultérieurement. La Fig.23 montre un exemple de cette sorte d'utilisation : un ficher **test.edir**, créé par la commande macro **slevent** (Fig.23a), est utilisé dans la macro **hsevt** comme fichier d'entrée pour produire un histogramme (Fig.23b). Les données sélectionées par **test.edir** pourraient aussi servir comme fichier d'entrée d'une sélection ultérieure (Fig.23b).

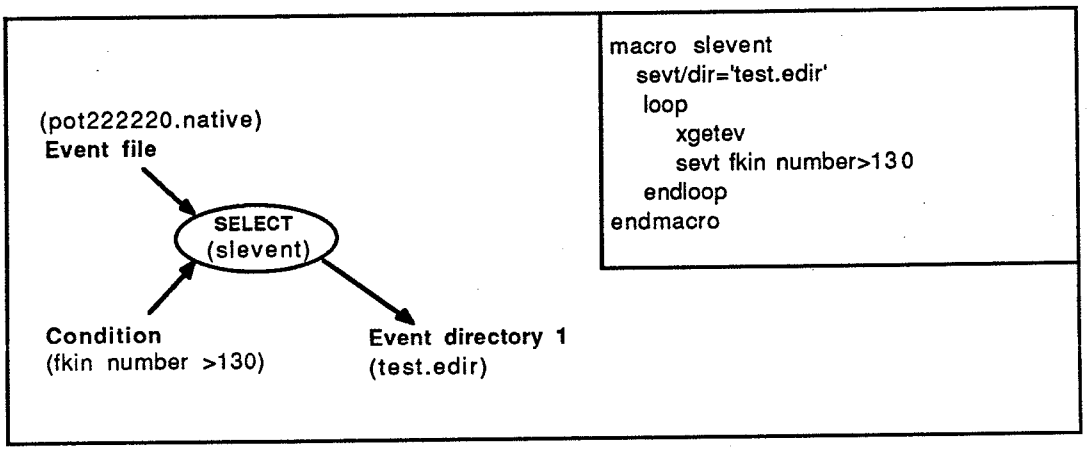


Fig. 23a La création d'un fichier sélectionné :
"pot222220.native" est le fichier contenant les données reconstruites, "test.edir" est le répertoire des événements de ce fichier.
à gauche : la procédure "select" sélectionne les événements avec le critère physique "condition" et écrit dans le fichier "event directory 1" les informations sur les événements satisfaisant à "condition"
à droite : la macro effectuant cette sélection

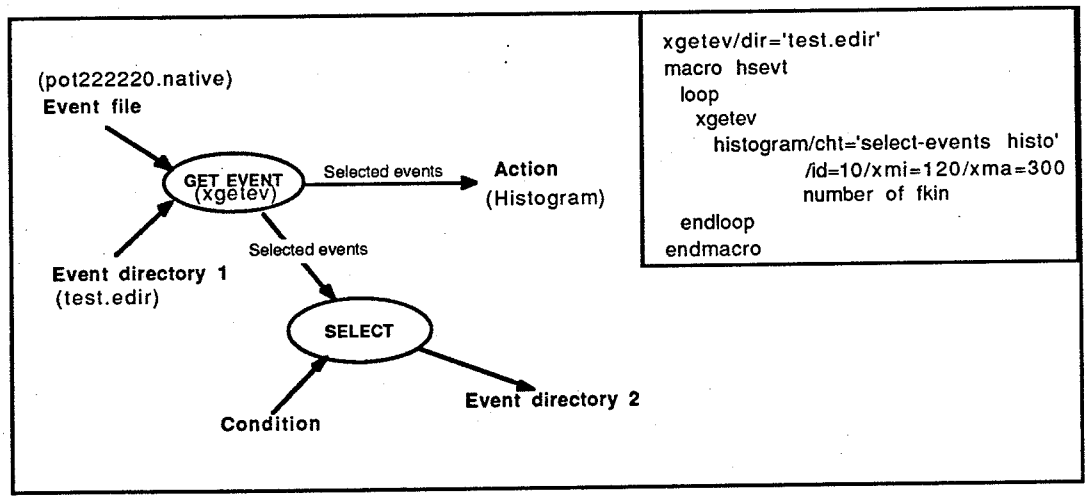


Fig. 23b L'utilisation du fichier sélectionné :
à gauche : la procédure "get event" lit les événements en se référant à "event directory1", les événements sélectionnés peuvent être utilisés directement par "action", ou utilisés comme la source de la procédure "sélect" pour une sélection supplémentaire
à droite : la macro effectuant ce travail

Réalisation

La base de ce mécanisme est le sélecteur **dsel**. Un événement satisfait à certaines conditions, si **dsel** contient au moins une ligne (page 49). Un ensemble de subroutines réalise cette fonction. On crée ainsi un répertoire d'événements dans lequel sont écrites les références des événements (nom de fichier original, numéro d'événement, numéro de run , index d'événement) satisfaisant aux conditions de sélection.

3.4 Commentaire

L'utilisateur dispose d'une vue complète d'un événement : la représentation d'événement, les données, les relations entre objets, la représentation statistique (histogrammes), etc.

Certains points faibles du système ont été remarqués pendant le développement et la maintenance :

- Le système utilise le logiciel de reconstruction (JULIA) pour accéder aux événements et pour effectuer la reconstruction interactive. Vu que JULIA lui-même est un système en évolution, l'intégration de JULIA dans PIGAL cause des difficultés certaines dans la maintenance du logiciel.

- De plus en plus de subroutines FORTRAN ont été ajoutées pour la représentation d'événement, le système est devenu gros et compliqué.

- L'utilisateur a besoin de quelques fonctionnalités nouvelles : la représentation d'histogramme bidimensionnel en graphique 3D, la représentation des triggers d'ALEPH (TPC, ITC, ECAL, HCAL...), l'intégration de certaines fonctions du progiciel d'analyse physique d'ALEPH. Par ailleurs il existe des outils qui correspondent exactement à ces besoins, par exemple PAW pour la manipulation d'histogrammes et TRIGGER_DISPLAY développé dans le groupe de on-line d'ALEPH. Les deux systèmes ont été développés dans leur propres contextes, utilisant notamment des ressources différentes. PIGAL n'a pas le moyen d'en profiter puisque les trois programmes ne sont pas homogènes.

Pour qu'un système d'analyse interactive soit bien bâti, on a besoin d'une nouvelle conception pour :

- faciliter la maintenance du système
- intégrer les logiciels hétérogènes

C'est le point de départ d'un nouveau système basé sur une architecture répartie, qui sera décrit dans le prochain chapitre.

Chapitre 4

Une architecture répartie pour PIGAL

PIGAL permet d'accéder aux événements de différentes façons : la représentation graphique, les données, les relations entre objets, les histogrammes, etc. Le système est flexible, facile à utiliser. Mais il manque certaines fonctions nécessaires à une analyse physique complète. Pour répondre à ce besoin, certains logiciels développés séparément ont été considérés malgrés les problèmes d'incompatibilité (page 64).

C'est cette situation qui justifie de donner au système une architecture répartie. Celle-ci est maintenant réalisée dans le prototype présenté dans ce chapitre.

En plus des fonctionnalités déjà présentes, une telle architecture a permis d'intégrer la représentation de triggers, la représentation statistique des données, etc. L'environnement est réparti sur plusieurs processeurs et fournit l'interface multiple avec l'utilisateur.

Ce travail démontre la faisabilité d'utilisation de techniques distribuées dans cette domaine.

Dans les sections suivantes, quelques notions élémentaires sur le réseau d'ordinateurs, la communication et les systèmes distribués seront présentées, suivies par la description du prototype.

4.1 Notion générale du système réparti [33,34,35]

Un système réparti est une application composée de plusieurs sous-ensembles qui sont distribués sur un réseau d'ordinateurs. Pour caractériser un tel système, il faut décrire comment les ordinateurs sont mis en réseaux.

Réseau d'ordinateurs et Protocoles de communication

Un réseau d'ordinateurs est un ensemble d'ordinateurs et de terminaux reliés par des lignes de communication. L'utilisateur peut accéder à un réseau et partager avec d'autres utilisateurs les ressources matérielles ou logicielles : machines, périphériques, bases de données, etc.

On classe un réseau d'ordinateurs, selon la zone géographique qu'il couvre, en deux catégories : réseau local (**LAN** - Local Area Networks) et réseau à longue distance (**WAN** - Wide Area Networks). Un LAN est souvent décrit comme un réseau privé qui fournit une communication à haute vitesse. Le LAN le plus courant est Ethernet [54], utilisé largement dans LEP et ALEPH. Le débit maximal théorique d'Ethernet est de 10 Mégabit/seconde.

Les réseaux d'ordinateurs utilisent différents protocoles de communication. Un protocole est un ensemble de règles respectées par tous les composants du réseau pendant la communication, par ex. comment établir une communication, comment tester si l'information est transmise correctement, comment corriger les erreurs de transmission, comment rétablir la communication après une interruption, etc. Dans ce domaine, de plus en plus de constructeurs utilisent un modèle en 7 couches : ISO/OSI (International Standards Organization/Open System Interconnection). Ce modèle divise le problème de communication en 7 niveaux :

7 Application
6 Présentation
5 Session
4 Transport

3 Réseau
2 Liaison
1 Physique

en définissant un protocole pour chaque niveau. L'ensemble des protocoles (surtout les couches 1-4) garantit la communication et la coopération entre composants de réseau. Avec cette base, on construit les différentes applications.

Application répartie

Deux définitions sont utiles pour caractériser un système distribué.

- **Matériel réparti**
  Une architecture répartie se compose de plusieurs processeurs autonomes. Entre eux il n'y a pas de mémoire partagée. Les processeurs coopèrent à travers des réseaux de communication.

- **Logiciel distribué**
  Un logiciel distribué se compose de plusieur processus qui communiquent par l'échange de messages.

Les objectifs d'un système distribué sont :

1 **diminuer le temps de calcul** en utilisant le parallélisme
2 construire des applications **intrinsèquement distribuées**, par ex. système de courier éléctronique.
3 **assurer le fonctionnement du système** en cas de problème de matériel ou de logiciel. L'arrêt d'une partie du système n'entraine pas nécessairement l'arrêt de la totalité du système.
4 réaliser des applications qui utilisent des **services spéciaux**. Chaque service peut utiliser un ou plusieur processeurs spécialisés pour obtenir bonne performance et fiabilité. Si une nouvelle fonction doit être ajoutée ou si une fonction existante a besoin d'une puissance supérieure de calcul, il est facile d'ajouter de nouveaux processeurs.
5 **diminuer le temps de développement** d'un grand système en utilisant des modules ou des systèmes existants.

Dans le cas de PIGAL, l'intérêt actuel porte sur les trois dernières sections.

4.2 Programmation distribuée

Les applications réparties sont réalisées à l'aide de la méthode de la programmation distribuée. La différence entre la programmation distribuée et la programmation centralisée est décrite ici, la notion de RPC (Remote Procedure Call) [36] - un modèle de communication choisi pour le prototype est aussi presentée.

La programmation distribuée se distingue de la programmation centralisée par trois points :

- l'utilisation de processeurs multiples
- la coopération entre processeurs
- la possibilité de détecter et de récupérer les erreurs du système

L'utilisation de processeurs multiples

Le système se compose de différents sous-ensembles qui s'exécutent en parallèle sur des processeurs différents. La répartition des programmes est soit transparente, définie par le compilateur et par la bibliothèque d'exécution du langage, soit programmable, sous contrôle de l'utilisateur.

La coopération entre processus

La coopération demande deux types d'interaction : communication et synchronisation. La communication entre processus se divise en deux catégories : **passage de message** et **partage de données**.

Passage de message : modèle pour une communication point-à-point

La communication met en jeu deux programmes nommés "Emetteur" et "Récepteur". L'Emetteur initialise l'interaction par envoi de message ou par appel à une procédure à distance. Le Récepteur reçoit la demande d'une manière soit **explicite** soit **implicite.** La manière explicite permet

au Récepteur de recevoir le message conditionnellement. Le Récepteur a le droit de refuser une requête, par ex. un serveur de fichier peut refuser une requête "ouvrir un fichier" si ce fichier est verrouillé. La manière implicite est absolue, c'est-à-dire que le message est reçu sans condition, le code d'exécution est appelé automatiquement.

Ce mode de communication est dit **monodirectionnel**. Certaines applications ont besoin de communication **bidirectionnelle**, surtout pour des applications ayant un modèle **client/serveur**. Le client demande un service au serveur et attend le résultat renvoyé par ce dernier. La Fig.24 montre les differentes configurations de ce modèle. On utilise couramment deux types de communication bidirectionnelle : le mécanisme de **rendez-vous**[33] et le mécanisme de **RPC (remote procedure call)** [36.37.38.39]. Ce dernier mécanisme est choisi pour réaliser le prototype.

En utilisant RPC, l'utilisateur a l'impression d'effectuer un appel normal à une sous-procédure, mais en fait le programme appelant (Local) et la procédure appelée (Distance) résident eventuellement dans deux processeurs différents. Quand un appel RPC est évoqué, le programme L est suspendu, un message contenant les arguments d'entrée est construit et envoyé à D. La procédure D exécute la requête et renvoie les arguments de sortie mis dans un message à L. RPC est une interaction synchrone. Le programme L doit attendre jusqu'au moment où il est sûr que le message est bien arrivé à son partenaire D. Ensuite, L et D peuvent travailler soit en mode séquentiel soit en mode concurrent. L'acceptation d'un appel RPC est implicite dans la plupart de cas. La réception est sans condition.

Partage de données : modèle de communication entre plusieurs processus

Les données sont accessibles par plusieurs programmes, tout se passe comme s'il s'agissait d'une mémoire commune à tous les processus. En fait, c'est le système d'exécution qui diffuse les données communes entre processus. Une couche de logiciel réalise la liaison entre le programme d'application qui a besoin de données partagées et les processeurs physiquement distribués qui n'ont pas de mémoire partagée.

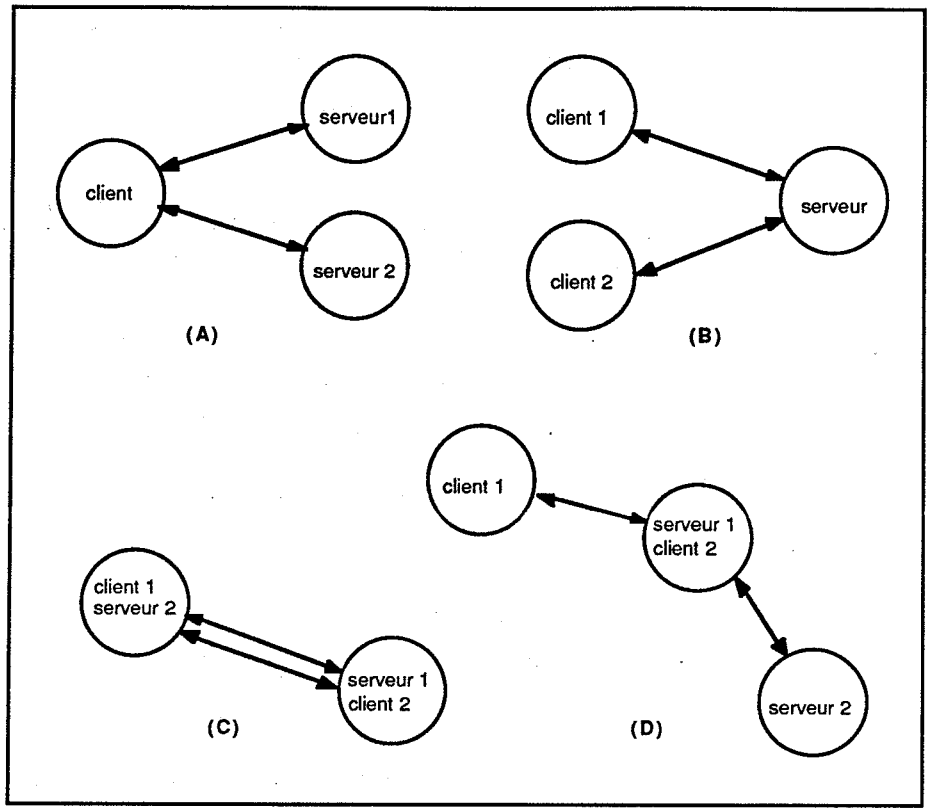


Fig. 24 un modèle Client/Serveur de communication bidirectionnelle :
(A) un client accède à plusieurs serveurs
(B) plusieurs clients accèdent à un seul serveur
(C) client 1 demande des services à serveur 1 qui utilise (comme client 2) certaines procédures situées en client 1 (comme serveur 2)
(D) client 1 demande des services à serveur 1 qui utilise (comme client 2) des procédures de serveur 2

La possibilité de détecter les erreurs et de récupérer le système en cas de défaillance partielle

Il existe trois méthodes pour traiter les incidents :

- **Transaction atomique :**
  Une application répartie consiste en plusieurs processus travaillant souvent sur un même objet (base des données, fichier partagé...). Pour garantir la cohérence de cet objet, on groupe certaines opérations ensemble en une **transaction atomique**. Une transaction est indivisible, elle est réussie si et seulement si chaque opération concernée est réussie.

Si un incident se produit (un processeur est tombé en panne, par exemple) pendant la transaction, les opérations concernées ont le moyen de retourner à leur état initial, comme si rien ne s'était passé. Cette propriété s'appele **récupérabilité**.

- **Tolérance transparente :**
  La sécurité du système est garantie par sa configuration. Elle est réalisée au niveau du système d'exploitation, par ex. deux processus identiques s'exécutent sur deux processeurs qui commutent automatiquement en cas de panne d'un processeur, ou bien un processeur central collectionne périodiquement les messages et les états de chaque processus utilisés pour la récupération.

- Programmation tolérante aux erreurs :
  le programme d'application surveille les processus détachés et récupère les erreurs en cas de problèmes.

4.3 Vue générale du prototype

Le système précédent (chapitre 3) a été divisé en plusieurs parties afin de pouvoir être réparti dans un réseau d'ordinateurs. Il n'y a pas beaucoup de changement du point de vue de l'utilisation, le système garde toutes les fonctionnalités d'origine. Par contre la représentation de triggers et la représentation statistique des données ont pu être ajoutées.

Toutes les fonctionnalités du programme sont regroupées dans 5 Serveurs qui portent les noms suivants :

- Serveur _Graphique : représentation d'événement en graphique 3D
- Serveur_JULIA : support de données
- Serveur_Histo : production d'histogrammes
- Serveur_PAW : représentation d'histogrammes en graphique 2D ou 3D
- Serveur_Trigger : représentation de triggers d'ALEPH

Pendant la phase d'initialisation du système, l'utilisateur peut placer dynamiquement les Serveurs sur les processeurs de son choix, par exemple l'on pourrait placer tous les Serveurs dans le même processeur (pour

faciliter le développement) ou alors les distribuer dans différents processeurs afin d'optimiser l'efficacité des Serveurs. En général, l'interface avec l'utilisateur sera regroupée sur un ou plusieurs écrans.

La Fig.25 est un exemple typique montrant les écrans de contrôle de cette sorte d'application. Elle représente trois écrans appartenant à une même session.

Au début de la session, l'utilisateur place les Serveurs désirés sur trois stations graphiques respectivement :

- Serveur_JULIA, Serveur_Graphique et Client résident dans le même processeur (Fig.25a)
- Serveur_Trigger fonctionne dans un processeur séparé (Fig.25b)
- Serveur_PAW fonctionne dans un troisième processeur (Fig.25c)

Le démarrage du système se réalise au fur et à mesure des besoins de l'utilisateur. Un Serveur est créé uniquement à la première demande d'un service qui en dépend. Donc les Serveurs dont aucun service n'est utilisé pendant la session ne sont pas chargés en mémoire. Ceci confère au système un maximum d'efficacité et de rapidité .

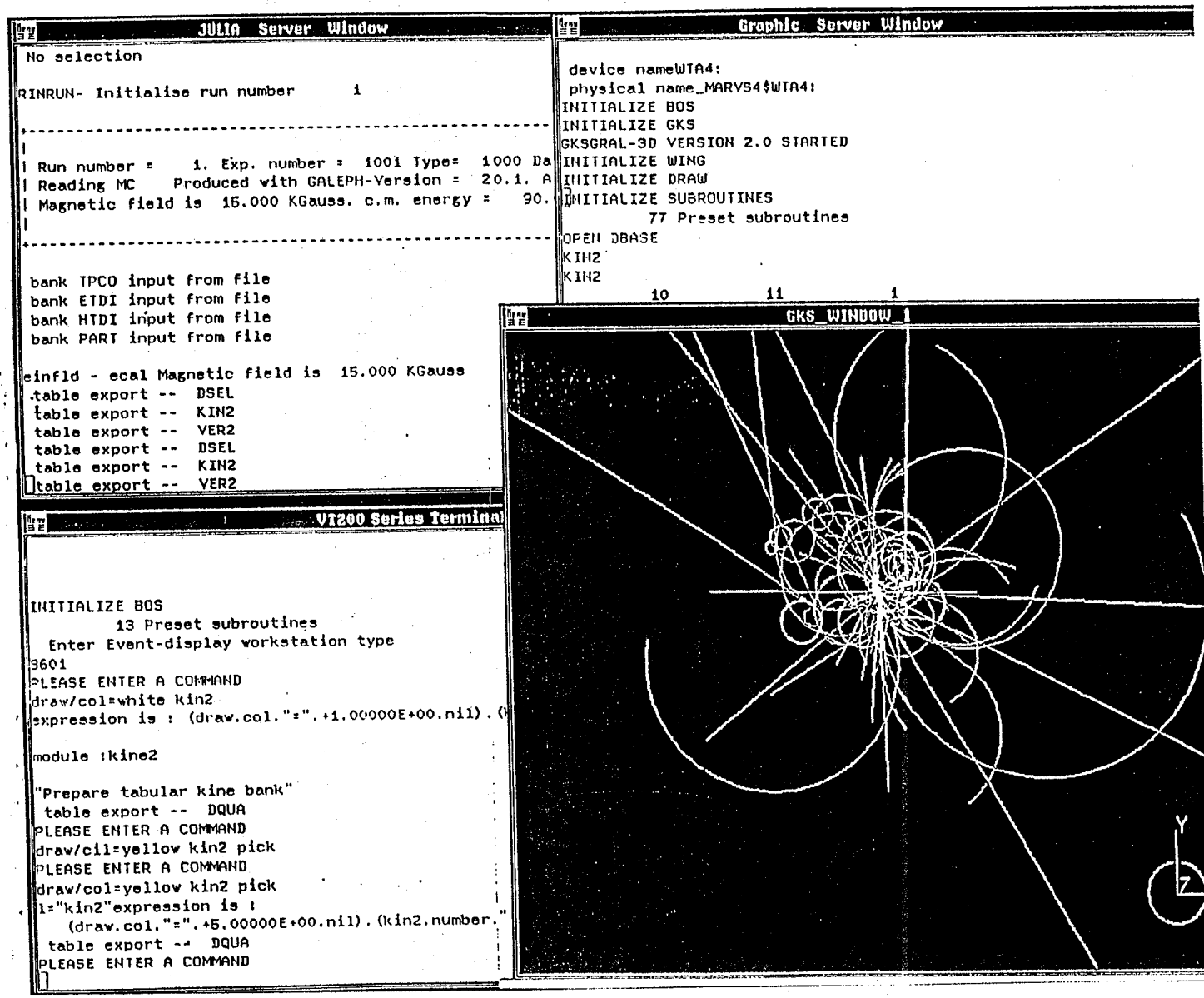
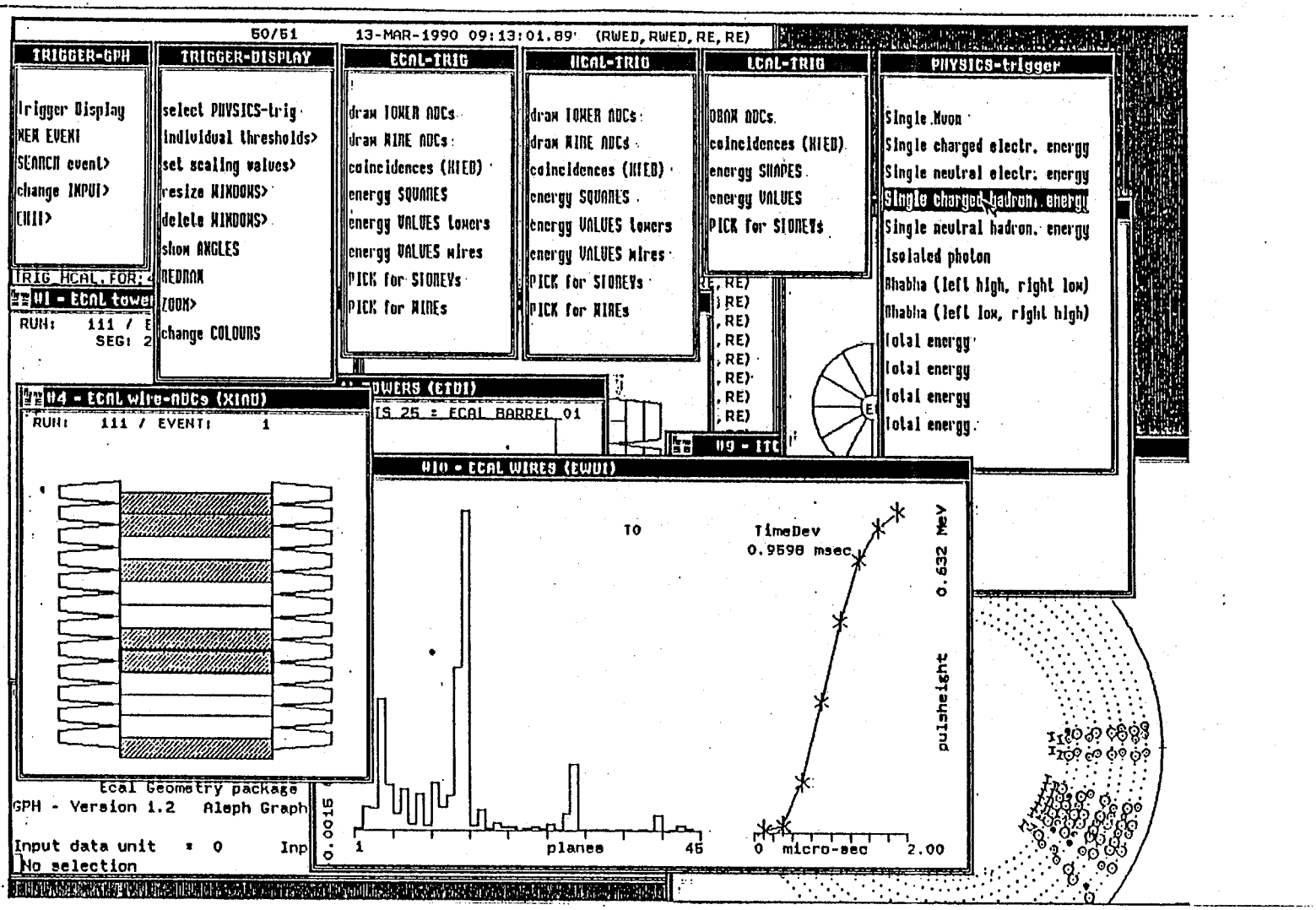
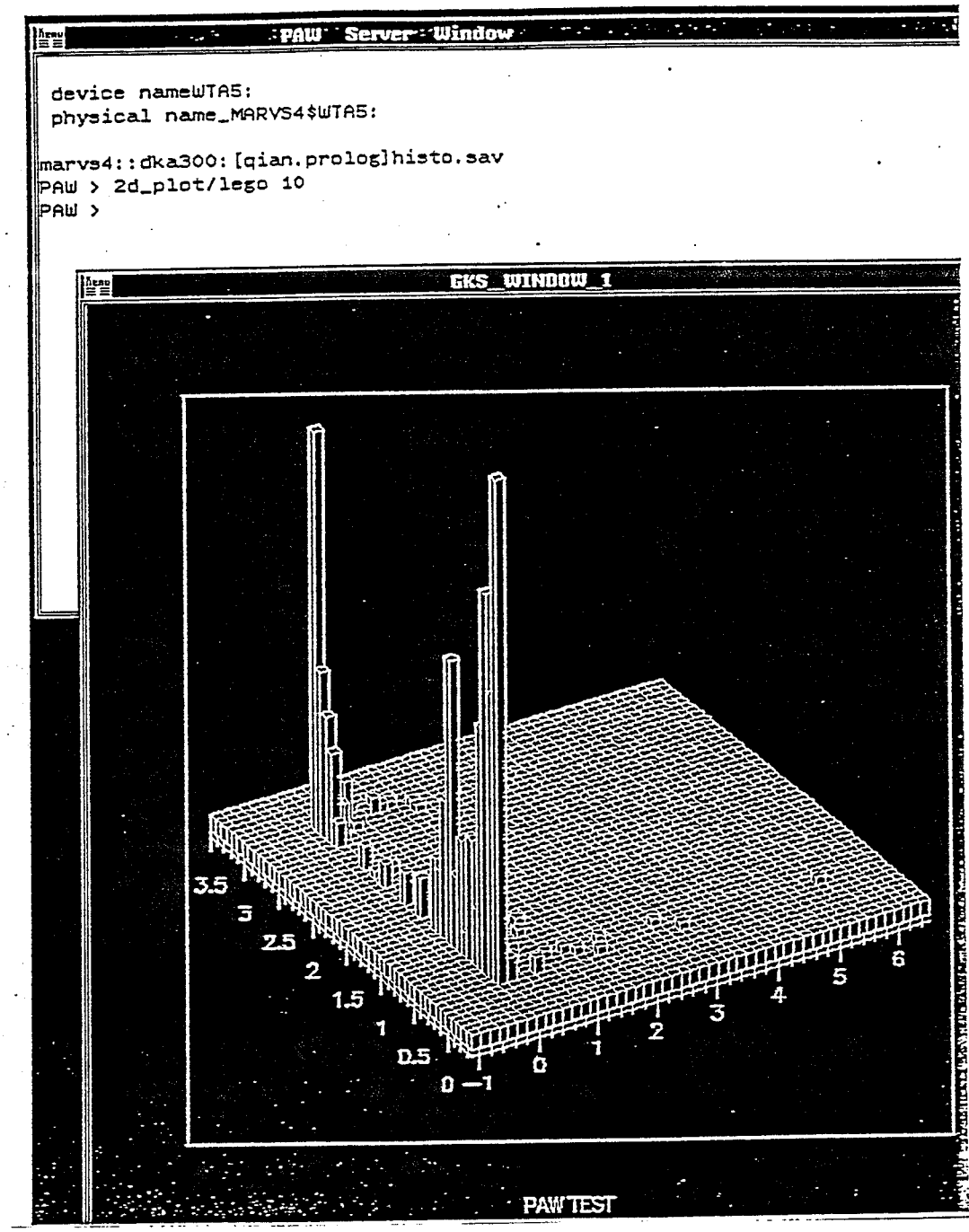


Fig. 25 Exemple d'un ensemble des interfaces d'utilisateur du prototype
a . (Ecran 1) Trois processus s'exécutent sur cette station : un Client (la fenêtre "VT200 series terminal"), deux Serveurs (les fenêtres "JULIA Server Window" et "Graphic Server Window"). L'image de l'événement est produite par Serveur_Graphique



b. (Ecran 2) l'écran de Serveur_Trigger



c. (Ecran 3) l'écran de Serveur_PAW
Cet histogramme est créé et rempli par PIGAL, représenté en mode "lego" par Serveur_PAW

4.4 Architecture du prototype

L'explication sur ce sujet sera divisée en deux étapes : le schéma logique du système suivi par l'architecture concrète avec la description de chaque composant.

Schéma logique

La structure logique du système est décrite Fig.26. Elle se compose de

- plusieurs serveurs qui effectuent chacun une fonction
- un pilote qui contrôle le système
- un mécanisme de transfert de commandes du pilote
- un mécanisme de transfert de données qui assure les échanges de données entre les serveurs (ou avec le pilote)

Le pilote contrôle et synchronise les serveurs par l'envoi de commande à chaque serveur. Une commande contient : le nom d'un service (action) et ses arguments.

Dans cette architecture il n'y a pas de relation serrée entre serveurs. Commandes et données sont transmises par deux mécanismes assurant l'échange d'informations. Leurs interfaces avec les serveurs et le pilote sont standardisées. Ceci permet d'obtenir une structure très souple. La synchronisation entre serveurs est assurée par le pilote. Celui-ci travaille comme s'il contenait une horloge interne.

Cette structure ressemble à l'architecture d'un ordinateur : les modules du programme sont l'équivalent des modules d'électronique, et les mécanismes de l'échange des informations remplissent le rôle des bus. Dans une famille d'ordinateurs du même constructeur, l'utilisateur peut choisir parmi les sous-ensembles disponibles, ceux qui sont nécessaires pour ses applications. Par la suite, il peut compléter sa configuration, ou bien remplacer certains éléments par de nouveaux modules, de performances supérieures.

Le présent travail montre qu'une approche identique est possible pour le logiciel. La modularité, dont les avantages sont reconnus depuis

longtemps dans le cas du matériel, est ainsi accessible dans le domaine du logiciel.

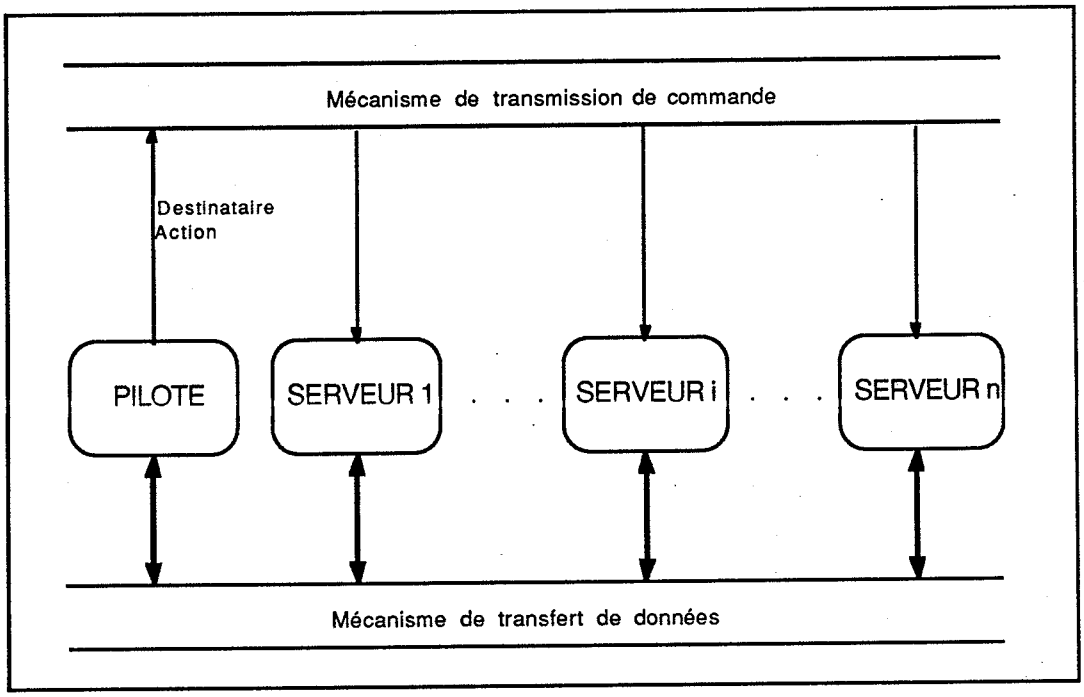


Fig. 26 L'architecture répartie (représentation logique).
Une série de serveurs liés par un mécanisme de transmission de commandes de contrôle et un mécanisme de transfert de données. Un pilote contrôle les serveurs par l'envoie de commandes.

Vue détaillée

Le prototype est basé sur le modèle Client/Serveur : le Client demande un service au Serveur et attend le résultat. Les différentes configurations de ce modèle sont déjà montrées dans la Fig.24, dont la configuration (A) a été choisie pour le prototype.

La structure du système est détaillée dans la Fig.27. On a 6 processus : le processus 1 est Client et les processus 2-6 sont Serveurs.

Le Client est le programme principal du système. Il analyse la commande de l'utilisateur, la transforme en une série de requêtes et envoie chaque requête au module Serveur concerné. Il se compose de deux parties, l'une est le programme principal de PIGAL, l'autre est le nouveau développement concernant le pilotage des Serveurs. Celle-ci

sera expliquée plus loin (page 83). Chaque Serveur a des fonctionnalités bien déterminées :

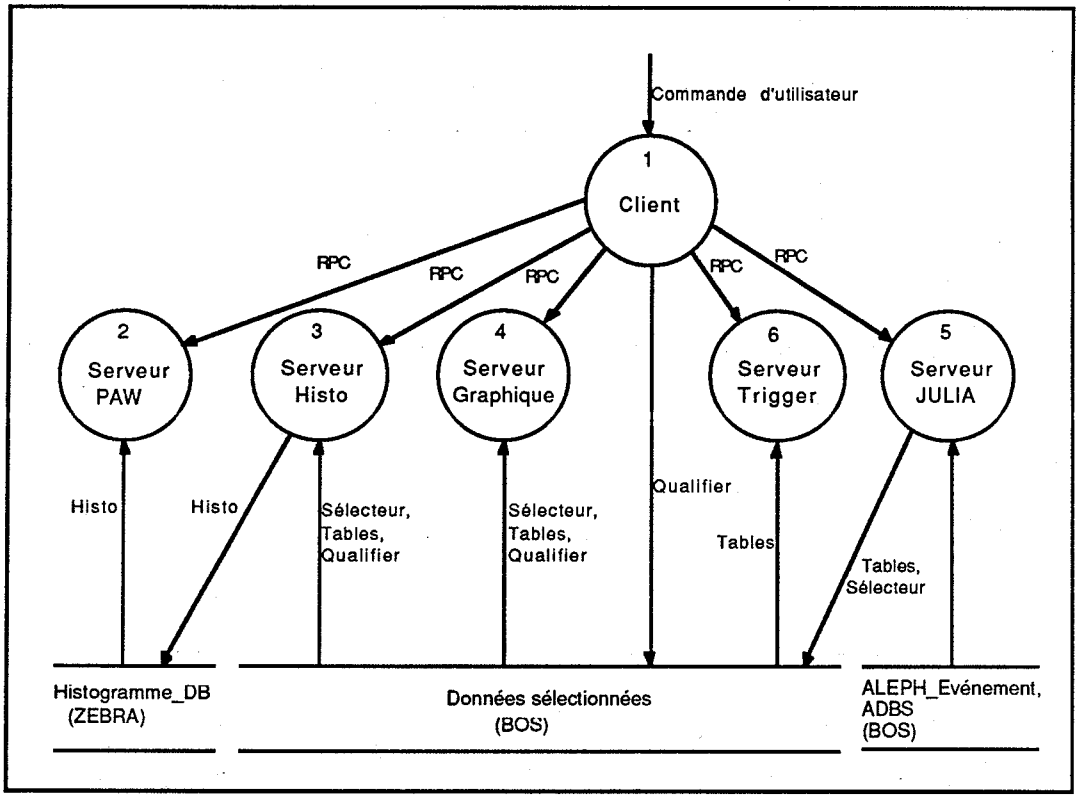


Fig. 27 Architecture multiprocessus :
Un Client, Cinq Serveurs,
Deux couplages :
1. le couplage de contrôle Client/Serveur réalisé en appel RPC.
2. le couplage de données Serveur/Serveur et Client/Serveur réalisé par Transfert de fichier

- **Serveur_PAW** (processus 2) :
  un sous-système qui représente les histogrammes en graphique 2D ou 3D. Le système original est PAW

- **Serveur_Histo** (processus 3) :
  un sous-système pour la production (création et remplissage) d'histogrammes. C'est le même module d'interface HBOOK que dans PIGAL (page 55).

- **Serveur_Graphique** (processus 4) :
  la partie principale de PIGAL, chargée de la représentation

d'événement en graphique 3D (page 37) et de l'impression de données sous forme de table.

- **Serveur_JULIA** (processus 5) :
  un ensemble de JULIA et SELTAB (page 46), qui accède aux événements et effectue les sélections éventuelles.

- **Serveur_Trigger** (processus 6) :
  un sous-système chargé de la représentation de triggers d'ALEPH (ECAL, HCAL, LCAL, ITC ...), développé à l'origine dans le groupe de l'acquisition de données.

Il existe deux niveaux de couplage entre les processus : couplage de contrôle et couplage de donnée.

- **Couplage de contrôle**
  Les contrôles s'effectuent seulement entre le Client et ses Serveurs. Il n'y a pas de relation de pilotage entre Serveurs. Les contrôles se réalisent par RPC. Le module Client envoie les messages au Serveur désiré et en reçoit les résultats (ou une notification d'erreur). Le format du message et l'interface RPC seront expliqués plus loin.

- **Couplage de donnée**
  Toutes les données échangées entre les processus (sauf Serveur_PAW) sont des tables entières qui respectent la structure definie par le dictionnaire DDL d'ADAMO. La structure des tables est indépendante de leur représentation physique. L'échange de données se réalise à travers un fichier sur disque. Ceci permet de construire un système indépendamment du gestionnaire de mémoire, qu'il soit BOS, ZEBRA ou C. Le couplage entre Serveur_PAW et Serveur_Histo est un fichier ZEBRA[40] pour des raisons de compatibilité avec PAW.

Les composants de chaque processus peuvent être très variés (voir la Fig.28). Ceci montre qu'une architecture distribuée permet de rassembler des ressources hétérogènes au fur et à mesure des besoins du développement.

PROCESSUS	1 Client	5 Serveur JULIA	4 Serveur Graphique	3 Serveur Histo	6 Serveur Trigger	2 Serveur PAW
Taille (kilooctets)	447	1140	1539	375	445	2195
Bibliothèques Appellées	PROLOG ALEPHLIB BOS RPC MOVE-TABLE	JULIA ALEPHLIB BOS RPC MOVE-TABLE	ALEPHLIB BOS GKS RPC MOVE-TABLE	ALEPHLIB BOS RPC MOVE-TABLE	ALEPHLIB BOS GPH-UIS UPI RPC MOVE-TABLE	GKS PAWLIB RPC

Fig. 28 Liste des bibliothèques appelées dans chaque Processus : RPC et MOVE-TABLE sont les sections assurant la coopération des processus

Dans le cas présent, l'hétérogénéité concerne :

- **les différentes versions des bibliothèques** : la plupart de programmes utilisent les bibliothèques ALEPHLIB et BOS, mais pas toujours dans la même version, car une nouvelle version peut impliquer la modification du reste du processus, ce qui ne peut être fait sans un travail conséquent.

- **l'incompatibilité des bibliothèques**. La représentation d'événements (processus 4) et la représentation des triggers (processus 6) utilisent des bibliothèques graphiques différentes (GKS, GPH-UIS).

Pour que ces différents composants puissent travailler ensemble, chaque processus intègre deux parties communes :

- la bibliothèque RPC, qui assure les **flux de contrôle** entre Client et Serveurs
- la subroutine MOVE_TABLE, qui assure les **flux de données** entre processus

Il faut mentionner ici que le regroupement des Serveurs est basé sur les données, c'est-à-dire que les subroutines qui travaillent sur les mêmes

données intermédiaires sont regroupées dans un Serveur pour réduire les communications entre Serveurs. Une autre façon d'organiser les Serveurs sera expliquée page 103.

4.5 Pilotage du système

Dans le prototype, le pilotage des Serveurs se réalise par les **opérations**. Une opération n'est effectuée que lorsque son résultat est nécessaire à une autre opération.

Voici un exemple simple pour aider à comprendre cette méthode. Le module Client recoit de l'utilisateur une commande

`draw track` (dessiner les traces)

Les analyses sont les suivantes :

analyse 1 : **draw** est l'un des services de Serveur_Graphique;
analyse 2 : la représentation de **track** nécessite l'exécution de la subroutine **rtrack** qui est aussi l'un des services du Serveur_Graphique;
analyse 3 : la table **track** est située originalement dans Serveur_JULIA.

Après l'analyse, le module Client soumet aux Serveurs les requêtes suivantes :

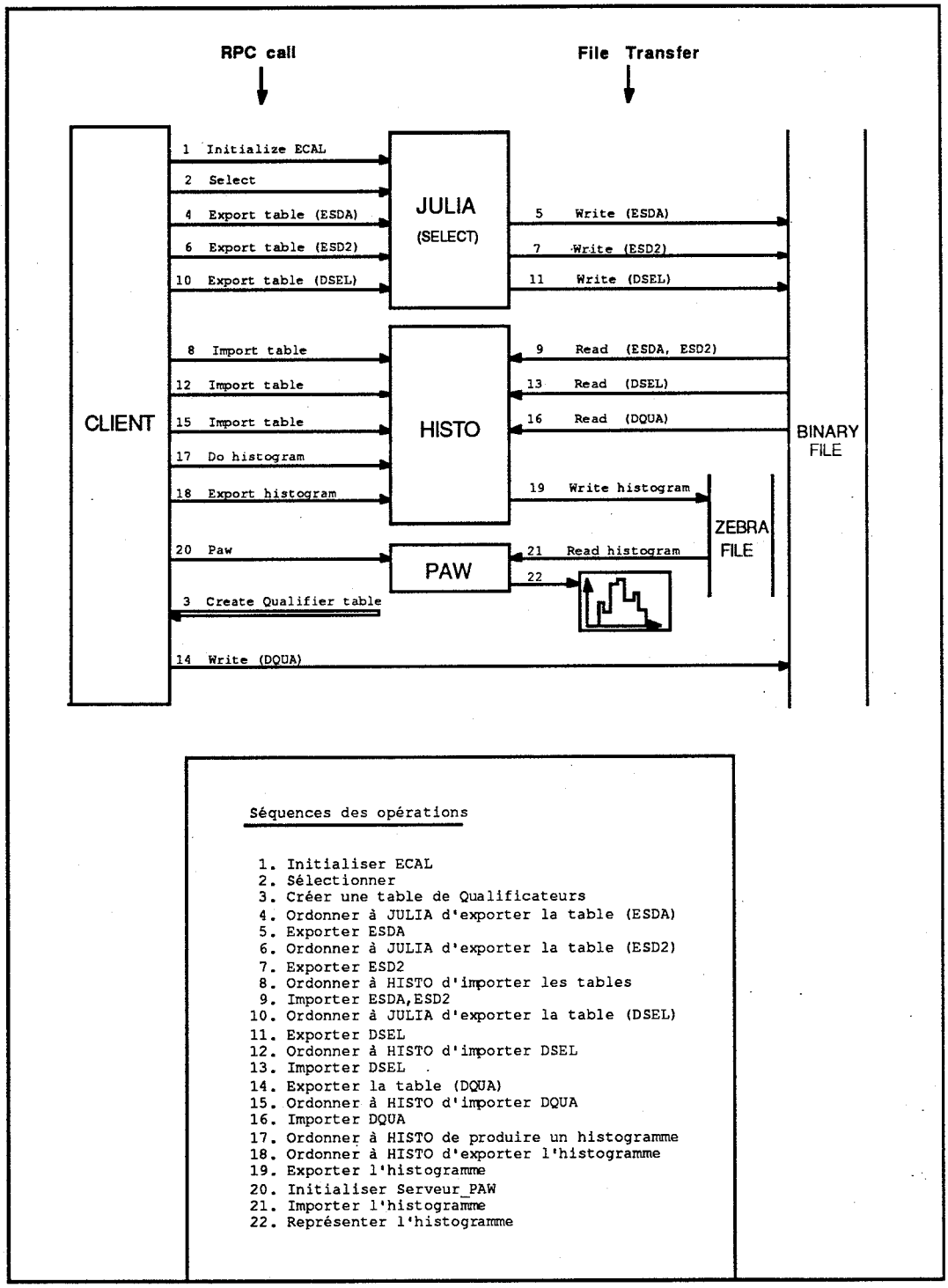
- Serveur_JULIA : exporter la table **track** dans un fichier binaire
- Serveur_Graphique : importer la table **track** depuis le fichier binaire et exécuter le code de la subroutine **rtrack** pour préparer les données
- Serveur_Graphique : exécuter la subroutine **draw** pour visualiser les traces.

Avant de soumettre une requête à l'un des Serveurs (JULIA ou Graphique), le module Client vérifie d'abord qu'il soit actif. Si ce n'est pas le cas, on l'active (voir en page 97).

La Fig.29 décrit la séquence des opérations concernant deux commandes :

```
histogram energy of esda
paw
```

La première commande crée et remplit un histogramme de l'énergie des cellules du calorimètre électromagnétique, la deuxième commande invoque Serveur_PAW (représentation d'histogramme). L'exécution des commandes concerne 4 processus (1 Client, 3 Serveurs) et crée deux fichiers (un binaire, un ZEBRA) pour la communication des données.



Séquences des opérations

1. Initialiser ECAL
2. Sélectionner
3. Créer une table de Qualificateurs
4. Ordonner à JULIA d'exporter la table (ESDA)
5. Exporter ESDA
6. Ordonner à JULIA d'exporter la table (ESD2)
7. Exporter ESD2
8. Ordonner à HISTO d'importer les tables
9. Importer ESDA,ESD2
10. Ordonner à JULIA d'exporter la table (DSEL)
11. Exporter DSEL
12. Ordonner à HISTO d'importer DSEL
13. Importer DSEL
14. Exporter la table (DQUA)
15. Ordonner à HISTO d'importer DQUA
16. Importer DQUA
17. Ordonner à HISTO de produire un histogramme
18. Ordonner à HISTO d'exporter l'histogramme
19. Exporter l'histogramme
20. Initialiser Serveur_PAW
21. Importer l'histogramme
22. Représenter l'histogramme

Fig. 29 Pilotage du système :
Ce schéma présente le mécanisme de pilotage du système et la synchronisation des Serveurs à travers un exemple de deux commandes de l'utilisateur : {histogram energy of esda} et {paw}. L'exécution concerne 1 Client et 3 Serveurs. Les séquences des opérations sont listées en bas du schéma.

4.6 Réalisation

Les éléments nécessaires pour réaliser le système sont :

- une base de connaissances pour piloter le système
- un outil pour réaliser les interfaces entre le Client et ses Serveurs
- un mécanisme pour passer les messages au niveau du contrôle
- un mécanisme pour le transfert de données entre processus et leur synchronisation
- un mécanisme pour les démarrages des Serveurs et les redémarrages après la réparation d'incident de Serveurs

Base de connaissances

Le module Client (en PROLOG) pilote les Serveurs à l'aide de sa base de connaissances. Elle contient les informations nécessaires pour gérer le système :

- **Pour chaque service :**
  La liste des entités nécessaires en entrée et en sortie, et le Serveur auquel il appartient. Des règles PROLOG adéquates permettent de répondre aux questions suivantes : étant donnée une entité, quel service doit-on exécuter pour la produire; quel Serveur doit-on initialiser pour exécuter ce service.

- **Pour la synchronisation du Serveur :**
  Une séquence d'opérations nécessaires pour transférer les entités d'un Serveur à un autre. Des règles PROLOG adéquates assurent la synchronisation entre Serveurs qui accèdent les même données. On active une opération avec la condition que l'opération précédente sur les même données soit terminée.

Les objectifs étant :

- de produire le moins possible de données répondant à la demande;
- d'activer le moins possible de Serveurs suivant le besoin.

Technique générale de RPC

Les interfaces Client/Serveur ont été définies et réalisées par RPC. Celui-ci est un modèle de communication (page 71), ainsi qu' une technique permettant de construire des applications coopératives. Les étapes suivantes sont nécessaires pour réaliser une application distribuée basée sur RPC :

- écrire les modules Client et Serveur comme s'ils allaient être assemblés directement
- écrire l'interface Client/Serveur en utilisant le Langage de description d'interface (Interface Description Language)
- compiler l'interface pour produire deux parties du code de l'interface només "**Stub**" : l'un pour le Client, l'autre pour le Serveur
- lier le module Client avec son **Stub** qui remplace le processus Serveur
- lier le module Serveur avec son **Stub** qui remplace le programme Client

Le compilateur du langage de description d'interface s'appelle **Générateur de Stub** (Stub generator). Il doit être compatible avec le langage utilisé par les modules Client et Serveur. Langage de description d'interface et Générateur de Stub sont des utilitaires de RPC.

La Fig.30 montre comment on peut appliquer cette méthode pour fractionner un système mono-processus volumineux en un ensemble de processus plus petits détachés les uns des autres.

L'on trouvera Fig.31 le schéma du mécanisme de RPC, les deux **Stubs** (client-**Stub**, serveur-**Stub**) chargés du codage et décodage d'arguments des appels RPC.

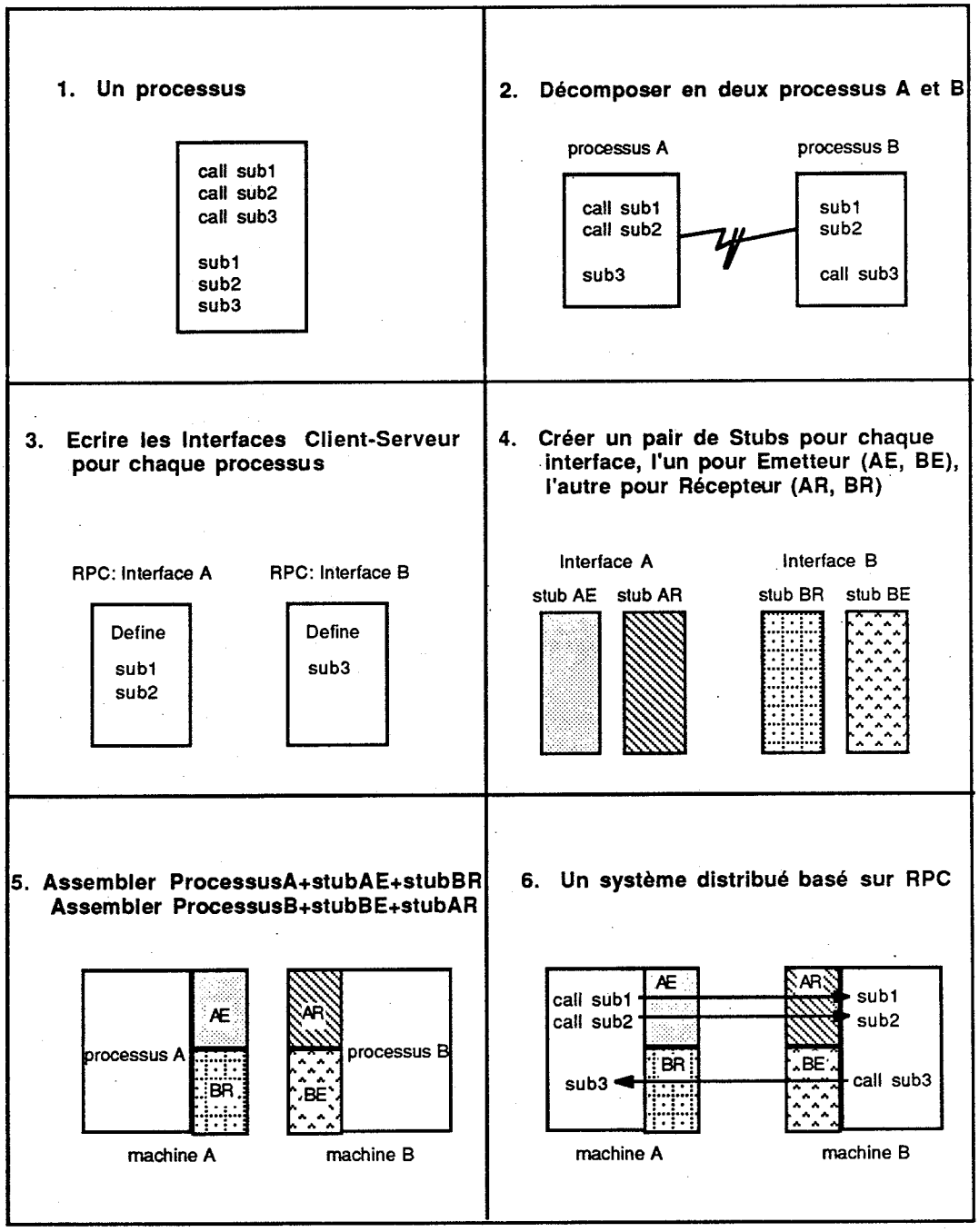


Fig. 30 Méthode pour fractionner un système mono-processus en une structure répartie en utilisant RPC

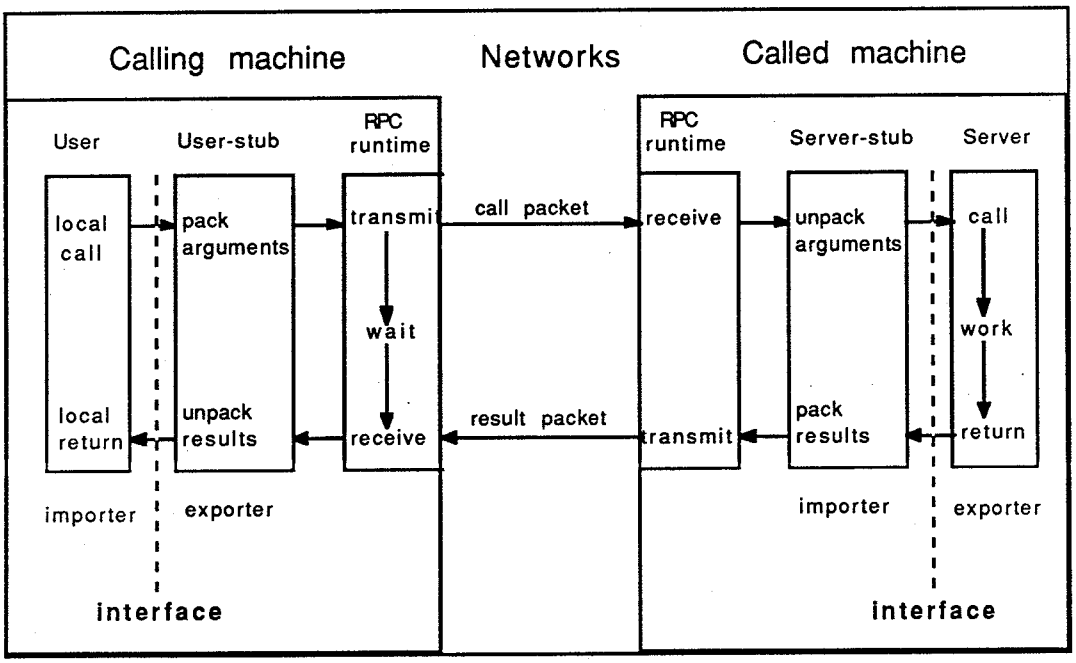


Fig. 31 Remote Procedure Call (RPC) [37] - un modèle de passage de message bidirectionnel. Le mecanisme de RPC (**Stub**, RPC runtime) gère la communication entre les procédures appelantes et appelées, qui résident dans deux processus différents.

Définition des interfaces Client-Serveur

Chaque fabricant d'ordinateurs, par ex. Digital, Apollo, Hewlet-Packard, fournit désormais son propre logiciel de RPC. A l'époque des tests, le système RPC développé au CERN était le seul disponible pour réaliser le prototype. Ce logiciel est développé pour les systèmes d'acquisition de données et aussi pour le contrôle d'expériences de physique des particules. Il s'adapte à plusieurs matériels et protocoles de communication, ainsi qu'à différents systèmes d'exploitation. La description détaillée de bibliothèque RPC se trouve dans le guide d'utilisateur [38]. Le logiciel contient :

- un langage (RPC Langage) pour la définition d'interface
- un précompilateur de RPCL qui transforme un fichier de définition ci-dessus en **Stub**
- une bibliothèque en temps réel qui assure la communication pendant

l'exécution du système.

A chacune des subroutines du Serveur que le module Client appelle pendant l'exécution correspond une procédure, déclarée dans le fichier de définition. On trouvera Fig.32 l'interface Client/Serveur_Trigger. La structure d'origine et les appels des deux processus sont donnés dans la section 1 de cette figure. La section 2 contient leur définition.

1. Les appels dans Client, les subroutines dans Serveur_Trigger

Client

```
character*80 cmd,tfile
•••
call t56(cmd)
call trig_dpl_init
•••
call trigger(tfile)
```

Serveur

```
subroutine t56(cmd)
•••
subroutine trig_dpl_init
•••
subroutine trigger(tfile)
•••
```

2. Le fichier de définition pour l'interface Client/Serveur_Trigger

```
PACKAGE rsubt IS

    TYPE  ch60     IS  string(60);
    TYPE  ch80     IS  string(80);

    PROCEDURE T56(
                C:   IN   ch80);
    PROCEDURE TRIG_DPL_INIT;
    PROCEDURE TRIGGER(
                C:   IN   ch60);

    PRAGMA CONCURRENT (TRIGGER);
    PRAGMA TIMEOUT (TRIGGER,400);

END rsubt;
```

Fig. 32 L'interface Client/Serveur_Trigger.
en haut : les appels et les subroutines situés à l'origine dans Client et Serveur
en bas : le fichier de définition de cette interface écrit en langage RPCL. La première instruction "PRAGMA" déclare que la procédure "TRIGGER" s'exécute en parallèle par rapport au reste du programme (voir Fig.36 & page 98). Ce fichier est compilé en deux **stubs** montrés Annexe 5.

Pour l'ensemble de PIGAL, cinq interfaces entre Client et Serveurs ont été définies :

Client / Serveur_Graphique
Client / Serveur_JULIA
Client / Serveur_Histo
Client / Serveur_PAW
Client / Serveur_Trigger

La complication des interfaces varie selon le niveau de couplage des processus. On trouvera Fig.33 deux exemples. L'un est la définition de Serveur_Graphique, l'autre est celle de Serveur_PAW. Le premier fichier est plus compliqué en raison du plus grand nombre de relations entre le module Client et Serveur_Graphique.

1. Le fichier de définition pour Serveur_Graphique

```
PACKAGE rsubg IS

   TYPE  inte      IS   rpc_long;
   TYPE  real      IS   rpc_real32;
   TYPE  ch4       IS   string(4);
   TYPE  ch80      IS   string(80);
   TYPE  ch500     IS   string(500);
   TYPE  iarr60    IS   array(1..2,1..30) of inte;
   TYPE  myarray   IS   array(1..2,1..50) of inte;

   PROCEDURE CALLSUG(
               INARG:   IN OUT   inte;
                CHAR:   IN OUT   ch500;
                IDES:   IN OUT   myarray);
   PROCEDURE IGETEV3;
   PROCEDURE WINPIC;
   PROCEDURE DRAW;
   PROCEDURE GTERMINAL(
                 CHAR:   IN       ch80);
   PROCEDURE SAVETYP(
                   N:    IN       inte);
   PROCEDURE GCOMMON(
                  N1:    IN       iarr60;
                  N2:    IN       ch500;
                  N3:    IN       inte;
                  N4:    IN       inte;
                  N5:    IN       ch4;
                  N6:    IN       inte);
   PROCEDURE PUTFLDG(
                  R1:    IN       real;
                  R2:    IN       real;
                  R3:    IN       real;
                  R4:    IN       real;
                  R5:    IN       real;
                  R6:    IN       real);

   PRAGMA CONCURRENT (IGETEV3,SAVETYP,GTERMINAL,GCOMMON,PUTFLDG,DRAW);

END rsubg;
```

2. Le fichier de définition pour Serveur_PAW

```
PACKAGE rsubp IS

   TYPE  ch80     IS   string(80);

   PROCEDURE PAW_MODIF;
   PROCEDURE PTERMINAL(
                CHAR:   IN       ch80);
   PROCEDURE KKUWHAG;

   PRAGMA CONCURRENT (PAW_MODIF,KKUWHAG,PTERMINAL);
   PRAGMA TIMEOUT (KKUWHAG,400);

END rsubp;
```

Fig. 33 Deux exemples de fichier de définition d'interface Client/Serveur écrits en langage RPCL: la complexité des interfaces varie selon les couplages Client/Serveur

Les 5 fichiers précédents, compilés par RPC, se transforment en 5 paires de **Stubs** en langage FORTRAN. Ceux concernant l'interface Client/Serveur_Trigger sont donnés dans l'Annexe 5 qui correspond à la Fig.32. Les 10 **Stubs** sont assemblés avec le Client et Serveurs respectivement (voir Fig.34), et établiront les liaisons entre eux pendant l'exécution du système. Leur utilisation est transparente et permet d'écrire l'appel aux subroutines en ignorant que le système est réparti.

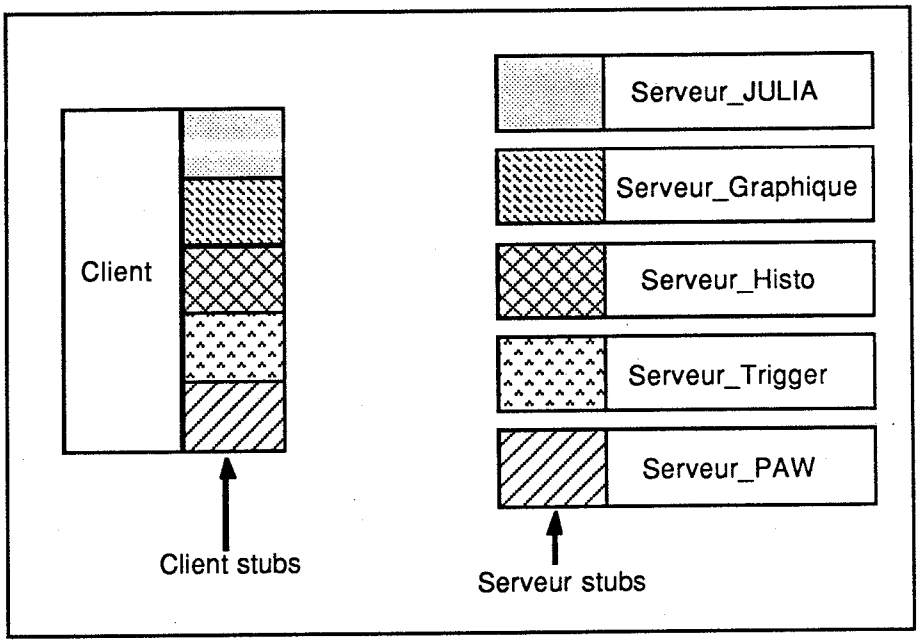


Fig. 34 Les interfaces Client/Serveurs du prototype présentées par 10 **Stubs** qui assurent les communications entre Client et chaque Serveur.

Echange de messages entre Client et Serveur

Chaque Serveur offre à l'extérieur une série de services connus par le module Client. Quand le Client a besoin d'un service, il envoie un message à un Serveur qui offre ce service. Ce mécanisme, appelé **échange de message**, se déroule comme suit :

- **Du côté Client**
  Comme on l'a vu dans la page 34, la plupart des appels à des subroutines FORTRAN passent par l'interface PROLOG-FORTRAN développée

pour PIGAL. Cette interface regroupe le nom du service et ses arguments dans un message et l'envoie au Serveur concerné par un appel RPC spécifique, déclaré dans le Stub de ce Serveur. Par exemple :

CALLSUJ (pour Serveur_JULIA)
CALLSUH (pour Serveur_Histo)
CALLSUG (pour Serveur_Graphique), voir la Fig.33 (page 92).

- **Du côté Serveur**
  une série de subroutines gère le décodage du message en deux parties : le nom du service d'une part et ses arguments d'autre part. Les procédures adéquates de la bibliothèque d'exécution assurent le branchement et font exécuter ce service (page 35).

- **L'exécution de RPC**
  Le système d'exécution de RPC assure la transmission de message vers le Serveur concerné.

On trouvera Fig.35a l'organisation d'un message, et Fig.35b un exemple complet d'appel de PROLOG à un Serveur. L'Annexe 6 décrit une partie du code de ce mécanisme avec les descriptions de chaque subroutine.

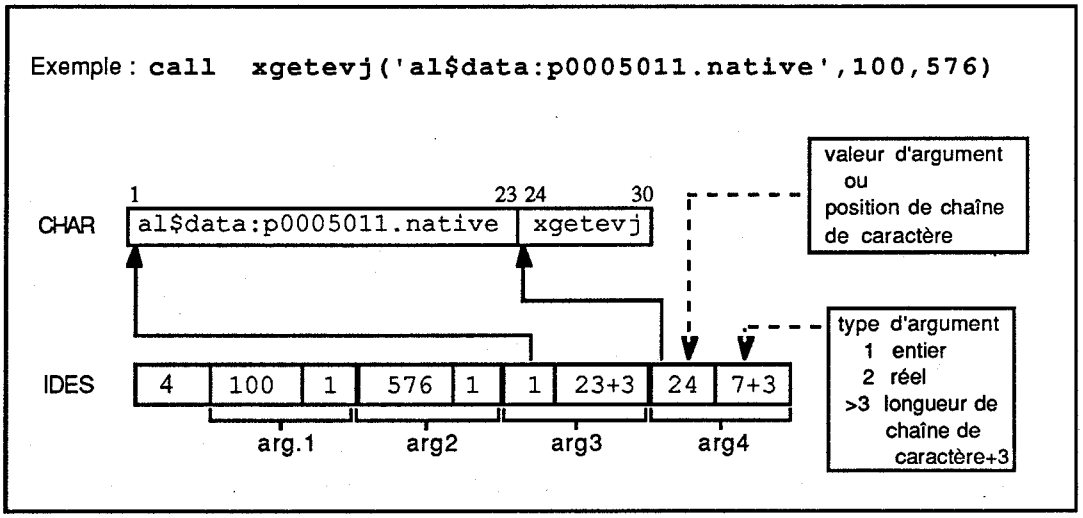


Fig. 35a Organisation d'un message : IDES est la liste d'arguments. Le premier mot de IDES est le nombre d'arguments (4 dans cet exemple). Ensuite un mot pour chaque argument: pour les arguments entiers ou réels, ce mot contient le valeur; pour les arguments de type chaîne de caractère, il contient un descripteur, le valeur de l'argument se trouve dans CHAR.

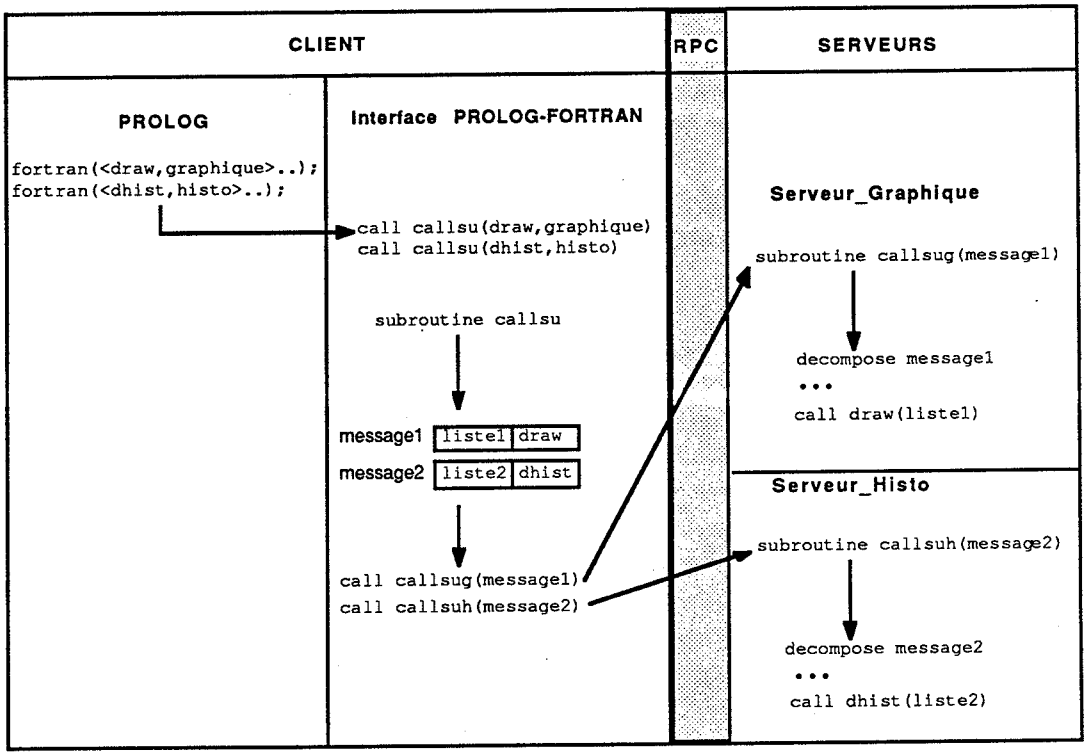


Fig. 35b Mécanisme de transfert de message.
Le module Client contient deux parties : PROLOG et l'interface PROLOG-FORTRAN où différents appels sont transformés en un seul appel de subroutine "callsu". Celui-ci forme les messages et fait appel à différents Serveurs. Chaque Serveur contient une procédure ("callsug" pour Serveur_Graphique, "callsuh" pour Serveur_histo ...) qui décode le message et appelle la subroutine adéquate pour exécuter le service demandé.

Transfert de données entre processus et leur synchronisation

Le transfert de données d'un processus à un autre est contrôlé par le module Client (voir la Fig.29). Par rapport à un bloc de données, on distinque un **Producteur** et un **Consommateur** qui sont tous les deux Serveurs. Un processus peut être le Producteur de certaines données et le Consommateur d'autres données. Pour cette raison, chaque processus intègre deux subroutines :

XOUTBK - écrit les données dans un fichier binaire en tant que Producteur;
XIMPORT - lit les données en tant que Consommateur.

La synchronisation entre Producteur et Consommateur est gérée par le module Client à l'aide de RPC. Les processus distribués basés sur RPC permettent de travailler en mode séquentiel ou en mode concurrent (voir la Fig.36). Pour assurer la cohérence des données communes, les deux appels RPC : "XOUTBK" et "XIMPORT" sont en mode séquentiel, c'est-à-dire que le Client ordonne au Consommateur d'importer les données uniquement quand il est sûr que le Producteur a déjà fini de les exporter.

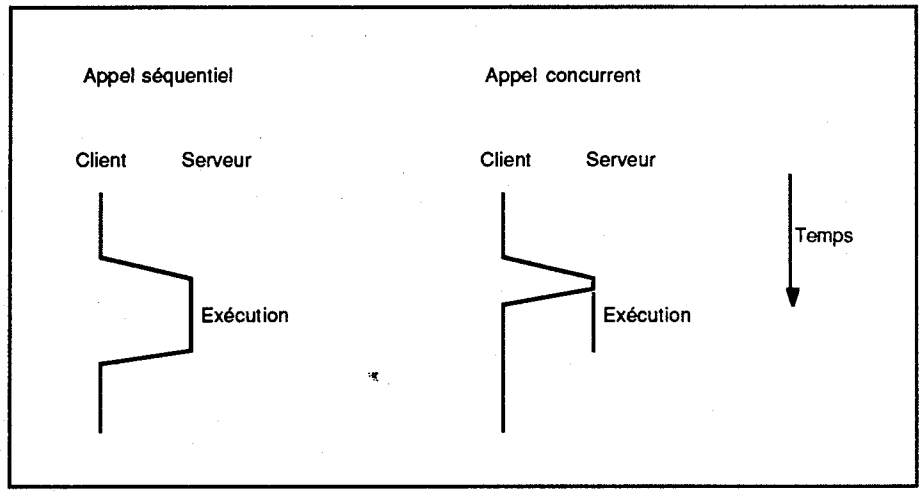


Fig. 36 Les flux de contrôle en mode séquentiel et en mode concurrent

Pour piloter le système, le module Client a besoin de savoir si le Producteur et le Consommateur résident dans le même processeur. Cette information (les relations entre le noms de processus et le nom de processeur) est déjà stockée dans le processus Client au moment de l'initialisation du système. Quand le module Client envoie un message à un Consommateur pour lui ordonner d'importer des données, le message contient l'information suivante:

- si le Producteur est dans une machine différente
- si oui, le nom de cette machine

Avec ces informations le Consommateur peut utiliser le support de données le plus adapté : mémoire commune ou fichier sur disque, avec ou sans adresse de réseau, etc.

Démarrage et redémarrage de Serveurs

Les utilitaires de RPC sont employés pour activer les Serveurs ainsi que pour les réactiver en cas d'incident.

Démarrage de Serveurs

Pendant l'initialisation du système, l'utilisateur attribue aux modules "Serveur" des processeurs physiques. Le système garde ces informations, mais aucun Serveur n'est vraiment activé à ce moment-là. Pendant l'exécution, lorsque l'utilisateur demande un service, le système vérifie si c'est la première demande concernant ce Serveur. Si oui, le Client active le Serveur par une commande :

```
call open_xxxx
```

où xxxx est le nom du **Stub** spécifié dans le fichier de définition de ce Serveur, par ex. **rsubj** est le nom du **Stub** pour Serveur_JULIA, **rsubg** pour Serveur_Graphique (voir Fig.33). RPC établit une liaison sur réseau entre Client et Serveur, et la maintient durant la session. Les Serveurs inutilisés pendant la session ne sont jamais chargés en mémoire.

Traitement de panne physique ou logique d'un Serveur

Dans un système centralisé, les causes de panne sont limitées aux problèmes de **matériels** et aux erreurs de **logiciels**; le système s'arrête dans la plupart des cas. Dans un système réparti, les problèmes peuvent avoir une 3^{e} cause : les pertes de **communication.**

En général, ces trois problèmes pourraient être résolus grâce à une propriété de système réparti : la tolérance aux fautes (page 73). Ici on utilise quelques facilités du logiciel RPC.

Quand un incident survient, le traitement est différent suivant le mode (séquentiel ou concurrent) dans lequel le Serveur est en train de travailler.

- **Mode séquentiel :**
  Le Client suspend son exécution en attendant le signal du Serveur. Si l'appel ne peut pas se terminer correctement, RPC retourne un message d'erreur au Client. Quelle que soit la cause du problème (panne de communication, Serveur déficient ...), le module Client interromp la liaison en utilisant la procédure RPC :

  ```
  rpc_close (...)
  ```

  et peut alors tenter de ré-établir la liaison après la réparation en utilisant la procédure RPC :

  ```
  rpc_open (...)
  ```

- **Mode concurrent :**
  Le Client et le Serveur exécutent leurs instructions en même temps, les deux sont indépendants. Si le Serveur subit un incident quelconque, aucun message de retour ne sera renvoyé par RPC. Dans le prototype certaines procédures sont définies en mode concurrent. On peut les trouver dans la Fig.32, définies par l'instruction :

  ```
  pragma concurrent
  ```

  En cas d'anormalie, le Serveur s'arrête, le Client continue sans le savoir, jusqu'au moment où le Client appelle à nouveau une procédure du Serveur et reçoit un message disant que le Serveur n'existe plus. Le Client ferme la liaison avec

  ```
  rpc_close (...)
  ```

  et la rouvre par

  ```
  rpc_open (...)
  ```

  Ce mécanisme permet de localiser les incidents de Serveurs et évite de paralyser totalement le système.

4.7 Evaluation des performances du prototype

On a vu les avantages liés à l'utilisation d'une architecture répartie. Son principal inconvénient est un ralentissement général (du moins en mode mono-processeur) dû à la transmission d'informations sur le réseau et dans le cas du prototype à la transmission de données par un fichier sur disque.

Différentes configurations du système ont été utilisées pour tester les performances et mesurer la consommation de temps. Les tests utilisent une même série de commandes (Fig.37a) qui composent une boucle pour représenter 9 événements en graphique 3D. Dans un environnement distribué, cette macro-commande exécute au total 1440 appels RPC au niveau du contrôle et 216 instructions de transfert de fichier (FORTRAN read, write) au niveau des données.

```
!*******************************************
!  timing test
!-----------------------------
macro timing
  local l_count
  let l_count = 1
  loop
     exitif l_count = 9
     xgetev
     draw tpco
     draw/col=red esda
     draw/col=red hsda
     draw kin2
     draw/col=green frft
     draw/col=white eslo
     draw/col=white hslo
     let l_count = l_count + 1
  endloop
endmacro
!*******************************************
```

Fig. 37a Les commandes utilisées pour le test :
les commandes composent une boucle pour représenter 9 événements en graphique 3D dans un environnement distribué. La représentation contient les contours de l'appareillage, les cellules des calorimètres et les traces dans le détecteur TPC.

Le test a été fait en 3 étapes (voir la Fig.37b) :

Etape 1 : exécution du système en un **seul processus** pour mesurer le temps d'initialisation du système et le temps d'exécution.

Etape 2 : exécution du système en **3 processus** qui sont affectés au **même processeur** pour estimer l'augmentation du temps d'initialisation (dépensée pour établir les liaisons DECNET) et l'augmentation absolue du temps d'exécution (nombreux appels RPC et transfert de données sur le réseau).

Etape 3 : exécution du système en **3 processus** qui sont affectés sur **deux processeurs** pour bénéficier de l'exécution simultanée des programmes permise par RPC appel concurrent.

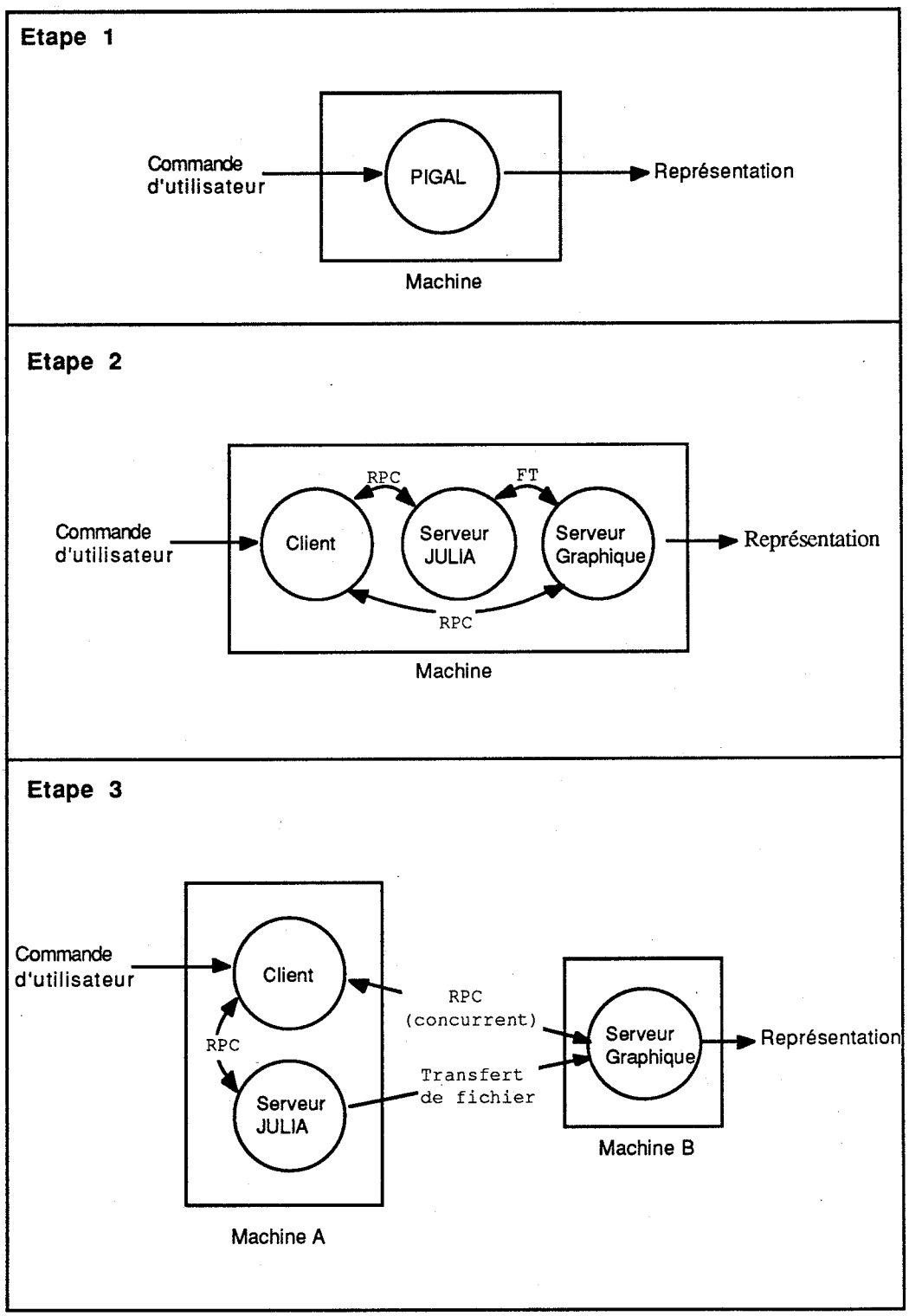


Fig. 37b Les configurations pour tester la performance du prototype :
étape 1 : exécuter PIGAL (mono-processus)
étape 2 : créer 3 processus dans une seule machine
étape 3 : créer 3 processus dans deux machines

Les résultats de ces mesures sont montrés sur la Fig.38. On peut noter deux points intéressants :

- le temps de communication est environ 24% du temps total d'exécution
- ce temps supplémentaire est compensé par l'exécution concurrente des processus quand ceux-ci sont répartis sur des machines adéquates

Dans le cas du prototype, Serveur_JULIA et Serveur_Graphique utilisent chacun une moitié de temps d'exécution total. C'est un bon équilibre permettant d'obtenir un temps d'exécution moindre que celui d'un seul processus (841 s contre 950 s).

Dans une application répartie, une partie importante de temps est consommée lors du transfert de données. On pourrait envisager l'utilisation d'une mémoire commune mais l'avantage de l'exécution en concurrence des Serveurs serait perdu parce qu'il implique le déroulement des Serveurs sur un même processeur.

	Condition d'exécution	Initialisation		Temps d'exécution des commandes		
		Temps Total	Temps pour établir les liaisons du réseau	Temps total d'exécution de macro-commande	Temps de communication sur Réseau Ethernet	Temps de comm. / Temps d'exécution
Mono-processus	VAX3100 8Mb Mem.	**39**	0	**950**	0	
Multiprocessus	**1 Processeur** VAX3100 8Mb Mem. 2 Decnet Link	**71**	32	**1262**	312	24.7%
	2 Processeurs 2 * VAX3100 2 * 8Mb Mem. 2 Decnet Link	**71**		**841**		

Fig. 38 Résultats de l'évaluation des performances du prototype (unité : seconde). Les chiffres mesurés directement sont indiqués en gras.

4.8 Expérience sur la décomposition d'un système

Cette section résume l'expérience acquise lors de cet essai d'adaptation d'une architecture distribuée à un système d'analyse de données de physique des particules.

Un programme existant ne peut être efficacement transformé en système réparti que s'il est fortement modulaire, avec un couplage faible entre les différents modules, ce qui était le cas de PIGAL dés le départ.

Les règles de la décomposition sont :

- Bien séparer les deux niveaux de couplage **contrôle** et **donnée** ; au niveau du **contrôle** : le protocole de messagerie doit être conçu soigneusement; au niveau des **données** : la structure de données doit être bien définie

- Le couplage entre processus doit être le moins serré possible, c'est-à-dire comporter le minimum possible de trafic de messages de contrôle et de données.

- Chaque Serveur est constitué d'un ensemble de procédures qui partagent soit les même données, soit les mêmes ressources logicielles. Avec le partage des données, chaque activité du système est exécutée de bout en bout dans un processus déterminé sans couplage avec un autre processus, à l'exception des paramètres d'entrée et de sortie. Cette décomposition minimise les communications et permet de bonne performance. En décomposant par ressource de logiciels, les parties utilisant les mêmes bibliothèques sont regroupées. Cette méthode risque de couper une fonctionnalité en deux et d'augmenter le trafic entre deux processus.

- Il faut aussi tenir compte des ressources matérielles du système. Il vaut mieux regrouper les parties utilisant les mêmes matériels.

Si l'on ne suit pas ces règles, le système sera compliqué et conduira à un couplage fort, avec de mauvaise performance. On trouvera Fig.39 trois décompositions possibles de PIGAL :

- **Configuration 1 :**
  Cette configuration contient le programme PIGAL entier, plus deux fonctions supplémentaires (représentation de triggers et représentation d'histogramme). Le temps d'exécution est long parce que le processus principal est trop grand avec beaucoup de fonctions mélangées.

- **Configuration 2 :**
  C'est une décomposition en ressource de logiciels. L'idée est de mettre ensemble toutes les subroutines utilisant la bibliothèque GKS, et former un module principal (en haut) avec l'interface utilisateur. Les subroutines de préparation sont restées dans un Serveur (au milieu) avec les autres subroutines, toutes utilisent les bibliothèques BOS et ALEPHLIB. Le résultat est que, après préparation, les données à transmettre à la couche graphique sont très nombreuses. Ceci implique une circulation importante entre processus (passée par appels RPC), ce qui ralentit le système. Dans cette configuration les données sont transmises par "callback", un mécanisme de communication (une procédure appelée par RPC peut faire un appel à une procédure située dans le module appelant). Un programme d'application utilisant la technique "callback" est difficile à mettre au point.

- **Configuration 3 :**
  C'est une décomposition en données. On a organisé les procédures autour de données communes. Dans cette décomposition partielle certaines fonctions ne sont pas encore divisées (dialogue, représentation d'événement, impression de données, histogramme), mais elle est déjà plus perfectionnée que les autres. En particulier peu de données circulent entre Serveur de reconstruction (au milieu) et le module principal (en haut). Le temps d'exécution est court.

  La comparaison de ces trois conceptions montre que la décomposition influence fortement la performance du système, notamment en ce qui concerne le temps d'exécution et de communication. De plus il faut faire attention à la complexité de l'interface entre processus : plus elle est compliqué, moins le système est fiable. Pour le prototype, les interfaces Client/Serveurs sont réalisées en sorte que le minimum nécessaire de procédures soit défini pour chaque Serveur.

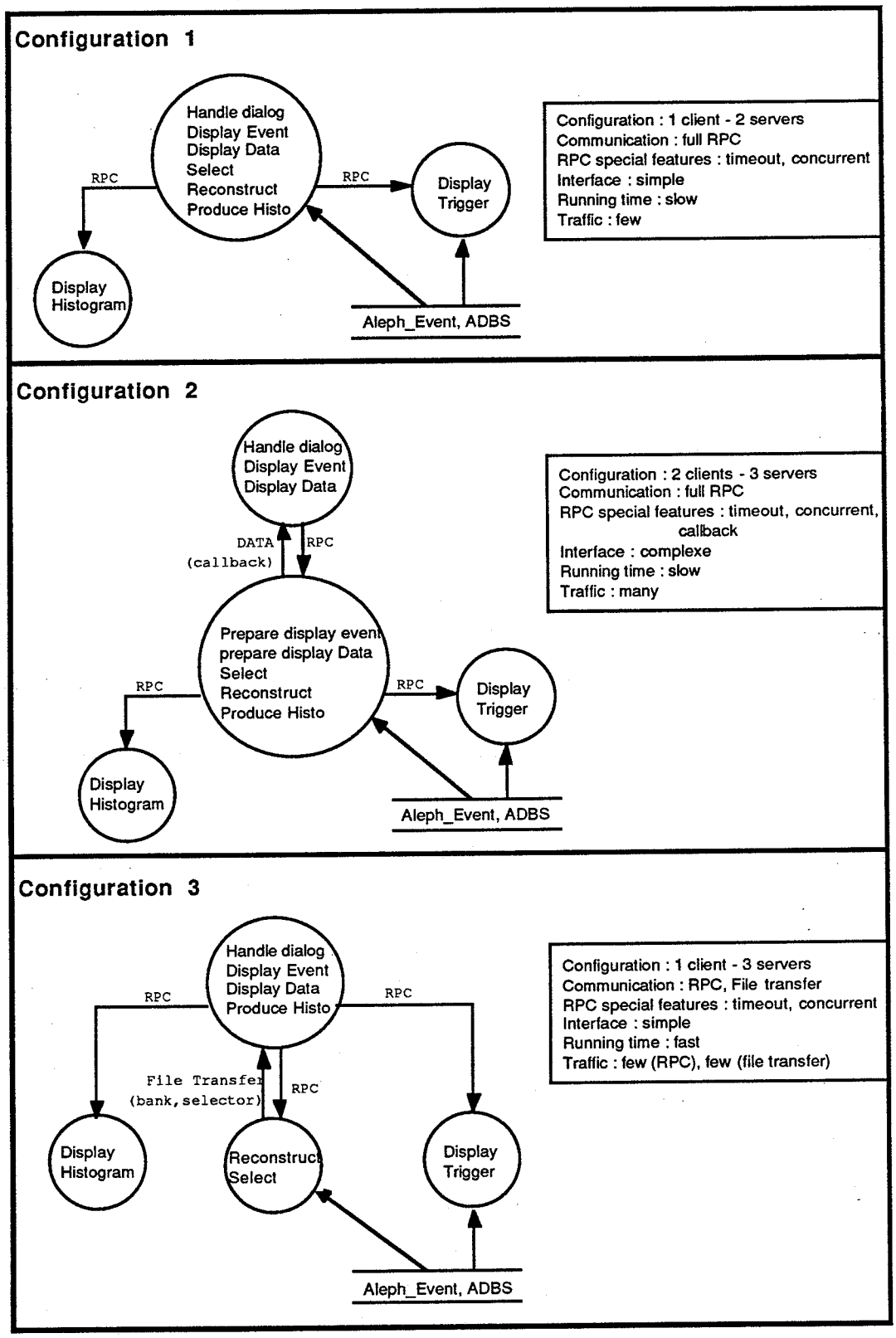


Fig. 39 Différentes configurations pratiquées pendant la décomposition de PIGAL

4.9 Améliorations éventuelles du système

Le prototype est loin d'être parfait. L'on peut envisager trois points à améliorer:

- **Le couplage de contrôle :** L'interface Client/Serveur n'est pas idéale, il se fait en fonction de l'ancien flux de contrôle de PIGAL. Changer ce flux de contrôle permettrait d'obtenir une interface beaucoup plus propre, par ex. les subroutines utilisées dans les initialisations de JULIA, GKS, BOS, PAW... sont originalement mélangées, changer les ordres de certains appels permet d'économiser le temps total de l'initialisation du système.

- **Le format de message** utilisé est simple. Il transmet un appel à chaque fois, ce qui augmente la charge du réseau. Il vaut mieux redéfinir le protocole ainsi chaque message contiendrait plusieur appels qui seraient considérés comme un seul service. Ce service serait considéré comme complet seulement quand tous ses appels seraient réussis.

- Dans la version présente, le système est bâti sur des machines de même type (VAX) et sur un même réseau local. Le module Client a suffisamment d'informations sur la configuration du système pour gérer le transfert de données (page 96). Il est possible d'ajouter d'autre information pour gérer une **configuration hétérogène** : différents types de machines et différents réseaux.

4.10 Conclusion sur la mise en oeuvre d'un environnement réparti

Avantage de RPC

RPCL(RPC langage) [38], langage permettant de définir l'interface entre les sous-systèmes, est facile à utiliser (voir Fig.32). Il offre une grande

facilité pour changer la configuration d'un logiciel sans toucher au code d'application.

Avantage de la structure distribuée

Le temps et le travail nécessaires au développement du système sont diminués

- chaque module est relativement petit, ce qui permet de localiser plus aisément les problèmes de logiciels.
- chaque module a une fonction homogène, ce qui permet de tester les modules séparément, de les mettre à jour séparément, d'ajouter de nouveaux modules sans toucher aux autres.
- il est possible d'utiliser un maximum de logiciels déjà existants, ce qui permet d'économiser un énorme temps de développement.

L'extensibilité du système est augmentée

La modularité du système autorise une extension souple et progressive du matériel et du logiciel en fonction des besoins.

La performance du système est améliorée

On pourrait affecter un processus à un processeur spécifique pour obtenir une meilleur performance. Par exemple :

- affecter le Serveur_Graphique à une station graphique de haute performance
- affecter le Serveur_JULIA à un processeur spécialisé en calcul numérique de grande vitesse pour effectuer le travail de reconstruction et surtout pour accéder rapidement aux données d'un événement

La fiabilité du système est augmentée

Le couplage de données entre processus est un couplage lâche. Chaque processus est autonome. Normalement, la défaillance d'un Serveur n'affecte pas les autres. Ceci permet d'éviter de paralyser totalement le

système en cas de panne.

Le temps d'exécution peut être diminué

Le temps d'exécution dépend du mode d'utilisation du système, ainsi que de la configuration de ce dernier. Quand on place un processus sur un processeur spécialisé, le temps d'exécution est déjà réduit. Si de plus l'on exécute les processus en mode concurrent, le temps d'exécution diminue encore. Le temps gagné compense largement les pertes dues aux communications sur le réseau.

Moyens et outils de test

Pour évaluer les performances d'un système distribué, on peut utiliser les logiciels d'analyse de performance fournis par les fabricants, par exemple PCA (Performance and Coverage Analyzer) [41] pour VAX. La méthode à suivre consiste à:

- Lier chaque module Serveur avec PCA en mode "DEBUG"
- Lancer séparément les Serveurs pour qu'ils puissent travailler en mode asynchrone
- Lancer le module Client

PCA doit collecter les informations de chaque Serveur permettant alors d'effectuer les analyses statistiques sur le temps d'exécution, le temps de communication, etc. L'utilisation de PCA dans un environnement réparti nécessite certains privilèges du système, malheureusement incompatibles avec l'utilisation partagée de la machine au niveau du laboratoire.

Choix de placement de processus

Placer de façon optimale un processus sur un processeur est un problème bien connu dans le domaine de l'informatique. Dans certaines architectures réparties, l'affectation d'un processus à un processeur se réalise au niveau du système d'exploitation. C'est par l'utilisation d'algorithmes bien étudiés que le système d'exploitation assure un meilleur équilibre entre le temps de calcul et le temps de communication.

Nous n'avons pas encore abordé ce problème avec le prototype, c'est l'utilisateur qui est le responsable de l'affectation des Serveurs.

Il est possible de faire mieux, par ex. si le module Client pouvait obtenir certaines informations statiques ou dynamiques sur l'état de chaque processeur : mémoire, nombre de processus exécutés à un moment donné, il pourrait aider l'utilisateur à choisir le processeur qui lui convient, par ex. le processeur le moint chargé. Certains articles concernant le problèmes de placement sont trouvés dans les références [42,43].

Langage de définition pour interface

Le logiciel RPC utilisé permet de construire des applications en langage mixte, par exemple Client en FORTRAN et Serveur en PASCAL, à travers l'utilisation spéciale du précompilateur du langage de définition RPCL [38]. Cependant pour construire un environnement hétérogène en tant que langage de programmation, il y a encore beaucoup à faire. Certains travaux concernant ce problème sont cités dans les références [44,45,46].

Chapitre 5

Changements récents et développements futurs

L'expérience acquise pendant le développement de PIGAL porte sur trois points principaux :

- visualisation d'événement en graphique 3D
- utilisation du langage PROLOG dans le domaine de l'analyse physique
- possibilité de construire un tel système sur une architecture répartie

Dans ce chapitre seront étudiés les évolutions et les développements futurs sur ces trois aspects, dont certains sont en cours de réalisation.

5.1 Graphique

Le graphique est la partie dont le changement est le plus fréquent dans PIGAL à cause du développement rapide des stations graphiques et de leurs logiciels. La modification la plus importante dans PIGAL a été de remplacer le noyau GKS par X-WINDOW [47] pour obtenir une représentation d'événements de haute performance tant pour la qualité graphique que pour la vitesse de représentation.

Le système X-WINDOW a été développé dans le projet Athena, au Massachusetts Institute of Technology. C'est un environnement logiciel

conçu pour traiter des applications graphiques sur des stations de travail. Il fournit aux logiciels d'applications une interface standard portable et une possibilité de développer des applications distribuées dans les réseaux locaux. Il contrôle la visualisation du graphique sur l'écran de la station de travail et l'entrée des données par la souris et le clavier.

Le système X-WINDOW est une bibliothèque de subroutines qui communiquent à travers un réseau. La bibliothèque se divise en deux parties : l'une pour les Clients, l'autre pour les Serveurs. Les interfaces d'application comprennent :

- un simulateur de terminal
- un gestionnaire de fenêtre
- des éditeurs de texte
- des interfaces pour graphique standard, par ex. GKS

La structure de X-WINDOW est presentée schématiquement dans la Fig.40.

La bibliothèque contient une série de subroutines graphiques élémentaires et de procédures pour manipuler les images en mode point (bitmap) gérées par les stations de travail. Ceci permet d'obtenir une représentation d'image à haute vitesse.

Le deuxième changement a été de se passer du système de gestionnaire de mémoire BOS au niveau du module graphique en le remplaçant par des appels à la gestion dynamique de mémoire inhérent au langage C. La combinaison de X-WINDOW et C donne au système une taille beaucoup plus réduite (1500 Kilooctets contre 4000Kilooctets), et une vitesse de représentation remarquable.

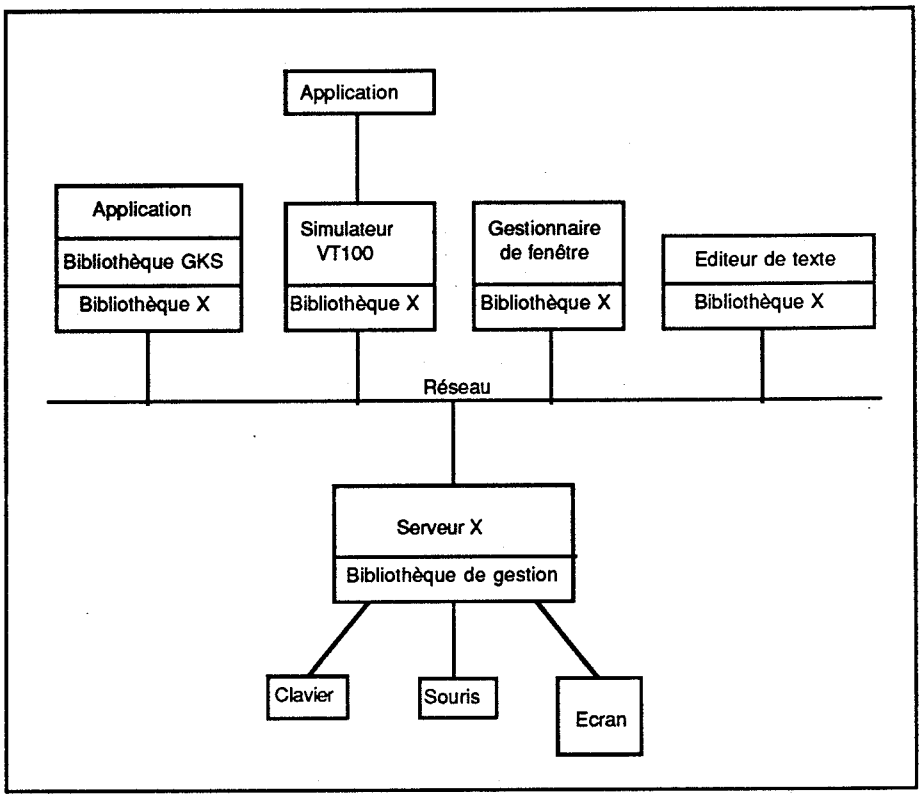


Fig. 40 Schéma logique de X-WINDOW :
Le système X-WINDOW gère l'interface entre l'utilisateur et sa station de travail, il fournit en outre un environnement standard aux programmes d'applications.

5.2 Prolog

Il manque à la version présente de PIGAL une série de commandes permettant de modifier les données sur lesquelles on travaille. On aimerait pouvoir effectuer des opérations du genre :

- ajouter une **colonne** à une table, lui donner un nom, et spécifier l'algorithme permettant de calculer les éléments de cette colonne. L'algorithme peut consister en opérations arithmétiques sur les colonnes existantes, ou bien faire intervenir des relations avec les autres tables.
- ajouter une **nouvelle table** à la structure de données présente et spécifier le mode de calcul de ses éléments.
- spécifier des **conditions logiques** permettant de sélectionner un

événement à visualiser.

Des solutions ont été apportées à ces divers problèmes [48]. Il est apparu qu'un langage de commande permettant d'exprimer de telles requêtes devrait être très complexe, donc difficile à analyser, mais également difficile à appréhender et à utiliser. C'est pourquoi un langage de programmation a du être développé. Etant donné qu'il s'agissait principalement d'exprimer des conditions logiques, il était normal de penser à utiliser PROLOG. De plus, pour PROLOG, le langage de programmation est également langage de commande, ce qui facilite grandement la mise au point des programmes d'utilisateur.

Le résultat est un environnement interactif, constitué de l'interpréteur de PROLOG et d'un système de vérification de contraintes. Quelques facilités ont été ajoutées pour la manipulation de données tabulaires. Cet environnement est caractérisé par :

- **l'accès aux table** sous forme de règles PROLOG, par ex.

  ```
  eclu(i,t,p,e)
  ```

  donne accès aux lignes de la table **eclu**, repérées par **i**, et dont les éléments sont identifiés par les variables **t,p,e**. La signification de ces variables a été spécifiée auparavant par la règle "define-view".

- la règle "define-view" permettant de **définir la partie visible d'une table**, par ex.

  ```
  define-view(eclu,index,theta,phi,energy)
  ```

- le **backtracking** (Annexe 3) **implicite** sur les lignes d'une table

- des opérateurs permettant de définir des **contraintes** sur les valeurs des variables en utilisant des opérations arithmétiques si nécessaire.

- la possibilité de **définir de nouvelles tables** avec leurs attributs et de les remplir ligne par ligne.

Pour conserver une vitesse d'exécution satisfaisante les données ne sont pas transférées dans le monde PROLOG. Au contraire c'est l'ensemble des règles (qui fait appel aux fonctionnalités décrites ci-dessus) qui sont exportées sous forme d'un arbre et exécutées par un programme récursif écrit en langage C. Le compilateur qui permet d'obtenir ce résultat est

écrit en PROLOG.

Ces méthodes d'analyse syntaxique et sémantique pourront être utilisées dans le contexte de PIGAL à deux niveaux :

- **Niveau global :**
  pilotage de la boucle de lecture des événements, sélection d'événements selon des critères logiques, pilotage de la partie graphique (cette partie est comme un sous-système, le système principal construit automatiquement ses commandes et les transmet à cette partie pour l'exécution).

- **Interne à un événement :**
  il permet de résoudre le problème de macros en leur donnant un véritable langage de programmation logique, avec accès à l'ensemble des données d'un événement.

5.3 Architecture logicielle répartie - généralisation

Les logiciels de "off-line" de la génération du LEP sont remarquables par leur taille, puisqu'ils comprennent entre 50000 et 100000 lignes de code. La qualité des logiciels, la durée du développement, le travail de maintenance sont les trois soucis majeurs des programmeurs. De plus en plus de stratégies et de techniques nouvelles sont appliquées depuis quelques années pour résoudre ces problèmes, notamment :

- La programmation orientée objets
- L'usage d'outils du type CASE (Computer-Aided Software Engineering)
- Le traitement distribué (Distributed Processing)

Cette dernière technique sera le sujet de cette section .

Changer la structure traditionnelle des logiciels de "off-line" (centralisée) pour une architecture répartie apporte les avantages discutés page 69 et testés sur le prototype du chapitre 4 (page 107). On

peut envisager une telle architecture pour d'autres applications.

Exemple 1 : **Le programme de reconstruction** devient en particulier nettement plus facile à mettre au point. Chaque Serveur est un logiciel de reconstruction entier pour un sous-détecteur, par ex. TPC. Le programme ne comporte pas de couplage serré entre différents Serveurs, évite les "common block", les entités intermédiaires de chaque Serveur étant exportées via le mécanisme de transfert de données (page 79) et étant importées par le Serveur suivant.

Exemple 2 : **Le programme destiné à effectuer l'analyse statistique interactive sur un lot d'événements** offrira à l'utilisateur les moyens d'appeler un logiciel d'analyse physique comme ALPHA (page 20), de regarder de temps en temps les événements particuliers en utilisant DALI (représentation d'événements en graphique 2D) ou PIGAL (graphique 3D) et de vérifier si nécessaire les données internes en utilisant le logiciel TIP (page 22).

L'intégration de ces nombreux logiciels préexistants dans un environnement unique pose deux problèmes principaux : d'une part l'incompatibilité des langages de commandes, d'autre part l'incompatibilité des structures de données internes ainsi que des systèmes de gestion de données utilisés. Ces deux problèmes pourraient être résolus au niveau de l'architecture du nouveau système en généralisant l'architecture utilisée pour réaliser le prototype du chapitre 4.

Une telle architecture, décrite dans la Fig.41, se déduit de celle de la Fig.26 en spécifiant qu'un module peut lui même être réparti (par exemple le cas du sous-système PIGAL). Dans ce schéma les deux types de **bus logiques** (Bus de commande et Bus de données) représentent respectivement le flux de contrôle (control flow) et le flux de données (data flow) de chaque niveau. Ils gèrent les échanges d'informations entre modules ou avec le moniteur du même niveau. Un **bus de commande** est défini par le protocole de communication et le format de message, un **bus de données** est défini par la structure de données et l' interface d'accès standardisée dans un système.

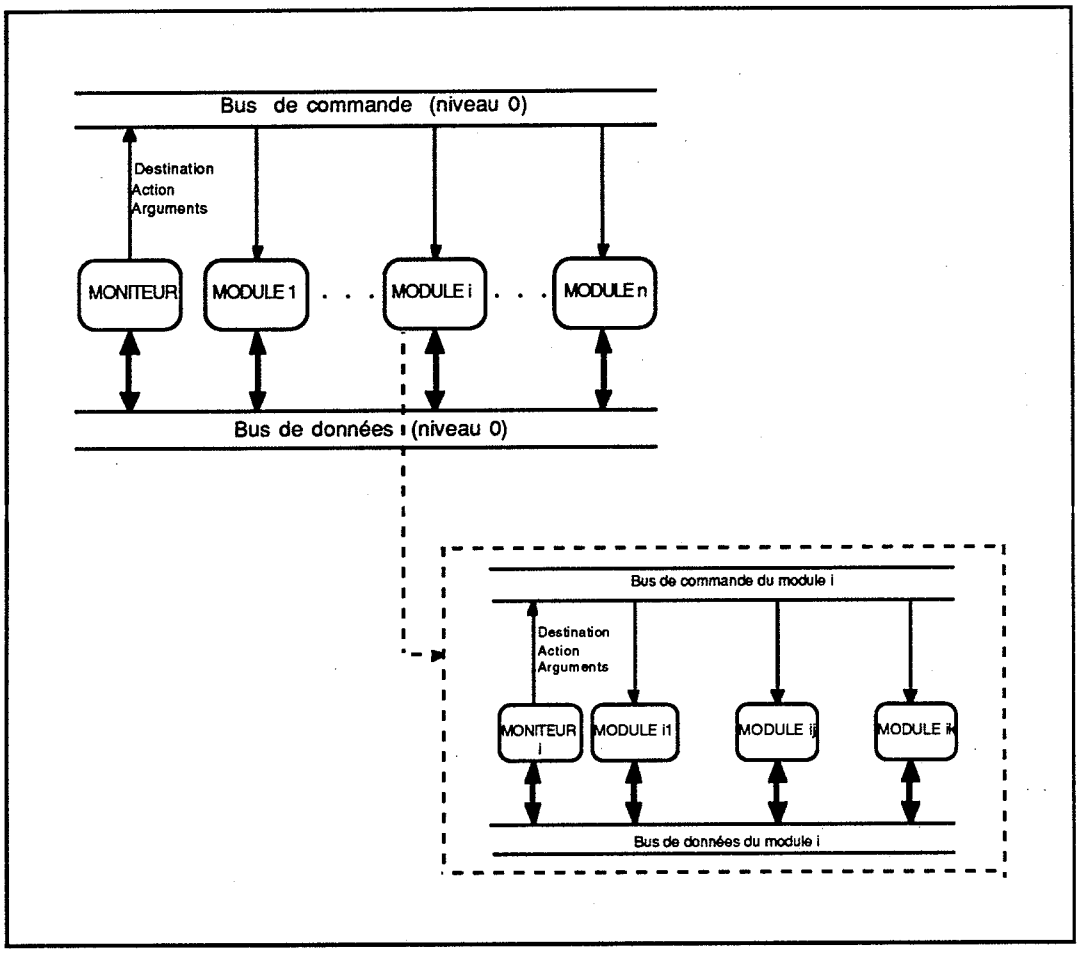


Fig. 41 Schéma logique d'une architecture répartie. Un niveau se compose de 4 parties principales:
1. plusieurs modules (Serveurs) effectuant chacun une fonction du système
2. un moniteur qui est le programme principal
3. un bus de commande assurant la communication entre moniteur et module au niveau de contrôle
4. un bus de données assurant l'echange de données entre modules (ou avec le moniteur)
Un module peut être un sous-système conçu de la même façon.

Les points importants dans cette architecture sont :

- Les bus de commande internes aux modules ne communiquent pas entre eux, ni avec le bus de commande externe

- Les bus de données interne aux modules ne communiquent pas avec les bus des autres niveaux

- Le moniteur du système principal ne transmet de commande qu'à ses propres Serveurs ou aux moniteurs des sous-systèmes, jamais directement aux Serveurs de niveau différent. Ainsi les commandes sont filtrées au niveau local, ce qui assure la sécurité. Les différents niveaux restent complètement autonomes et peuvent ainsi coexister malgrés des systèmes de gestion de données, des bibliothèques et des systèmes de commande à priori incompatibles.

Une idée similaire se trouve dans la référence[49] qui donne une notion de **"software bus"** et une description de **"self-describing system"**. Dans ce système, chaque bloc de données est un "objet", il contient une description sur soi-même et circule dans un "software bus". Les serveurs du système peuvent accéder à ces objets, lire leurs descriptions, et utiliser les données. Le projet est basé sur Unix System V. Le "common block" de langage FORTRAN et la **structure de données** de langage C sont les deux types permis par le "software bus".

Chapitre 6

Conclusions

PIGAL est basé originalement sur une architecture modulaire. Les modules principaux de PIGAL sont :

- l'analyseur grammatical de syntaxe
- la représentation interactive
- la sélection d'objets physiques
- la transformation d'objet physique en objet graphique 3D
- la manipulation de structures graphiques : représentation, vue, fenêtre...

Chaque action est réalisée par des subroutines FORTRAN spéciales et chaque subroutine n'effectue qu'une opération élémentaire. La séquence d'appel des subroutines FORTRAN est contrôlée par le programme principal en PROLOG qui comprend la base de connaissances sur le système.

La fonction la plus marquante dans PIGAL est son graphique interactif. L'image représentée fournit en fait une deuxième interface à PIGAL (l'autre étant l'interface en langage quasi naturel) à travers laquelle l'utilisateur peut piloter le système par l'intermédiaire de la souris.

Une nouvelle approche est expérimentée sur PIGAL : donner à PIGAL une architecture distribuée. Les modules élémentaires du système sont repartis dans un réseau d'ordinateur et communiquent à travers RPC et FT (transfert de fichier). Cette structure donne certains avantages importants au point de vue génie logiciel, permettant d'améliorer pas à pas un très grand système, de limiter les modifications, d'isoler les modules dans différents processus afin d'éviter les incompatibilités de ressources. La facilité de la reconfiguration dynamique de programme

permet d'utiliser le maximum de ressources disponibles à tout moment. C'est un point extrèmement important dans le développement des systèmes d'analyse de données vue la durée de développement et la rapidité d'évolution du matériel.

La construction de composants logiciels interchangeables et la réalisation de leur liaison standard sont les deux points critiques pour la prochaine génération de systèmes de logiciel. La séparation des flux de commande et flux de données est nécessaire. Un protocole bien conçu au niveau de commande et une structure bien définie au niveau de données garantissent les liaisons standards de composants logiciels.

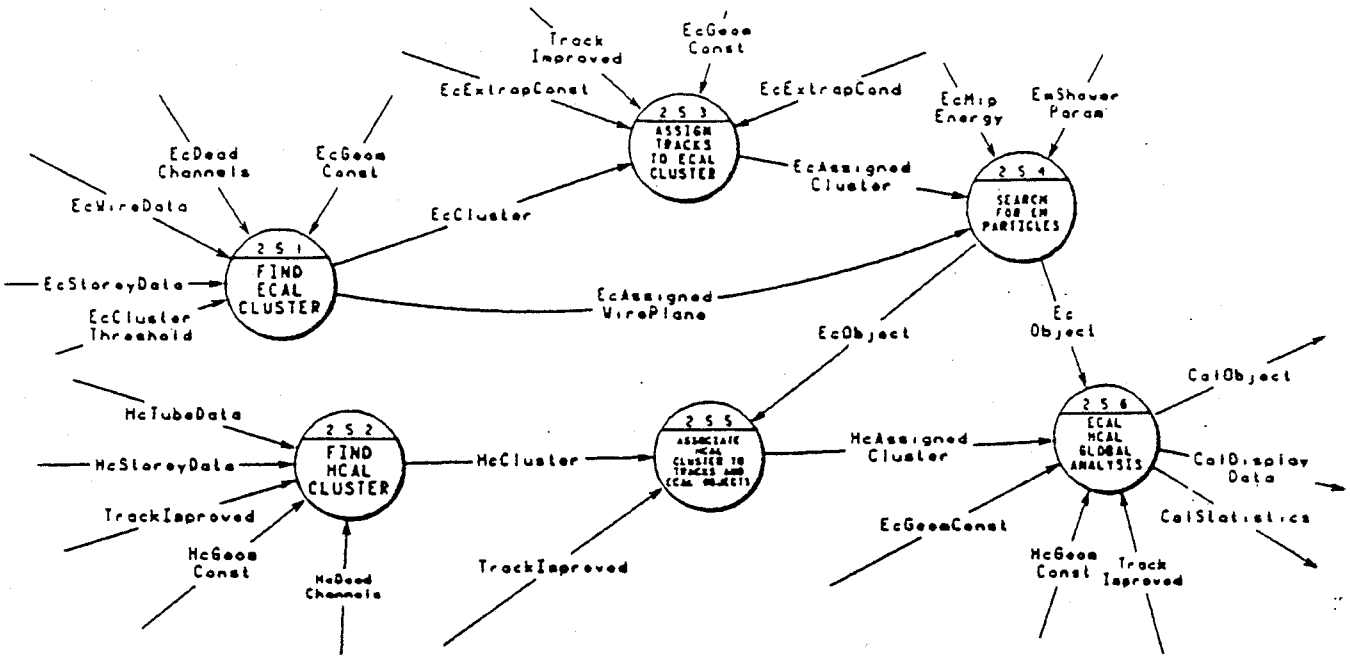
Le mécanisme de RPC (Remote Procedure Call) offre un outil puissant et flexible dans cet environnement.

Annexe 1

Exemples de DFD, ERD et DDL

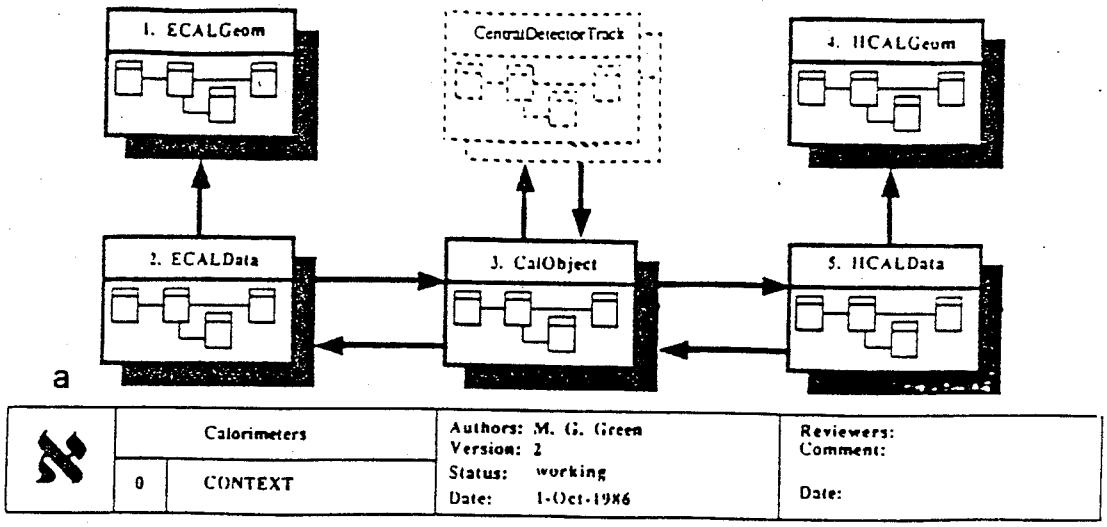
L'on trouvera ici les exemples de DFD, ERD et DDL extraits du document décrivant la reconstruction des énergies des gerbes dans le Calorimètre éléctromagnétique [15].

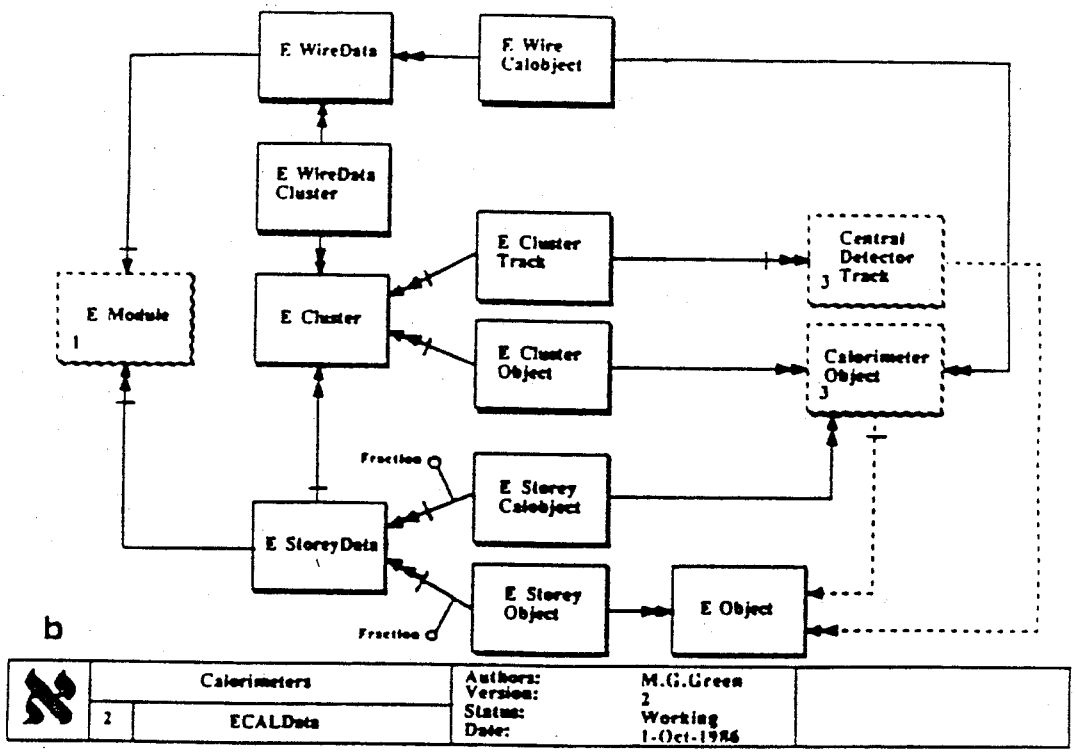
Le diagramme de flux de données



	Reconstruct Aleph Event		Author	Reviewer
			Version 1 0	Comment
	2 5	FIND CALORIMETER OBJECT	Status	
			Date 6-OCT-1986	Date

La premier ERD décrit la structure de données entière de ECAL (CONTEXT). La seconde décrit la structure de ECALData qui correspond à la deuxième boîte de la figure précédente.





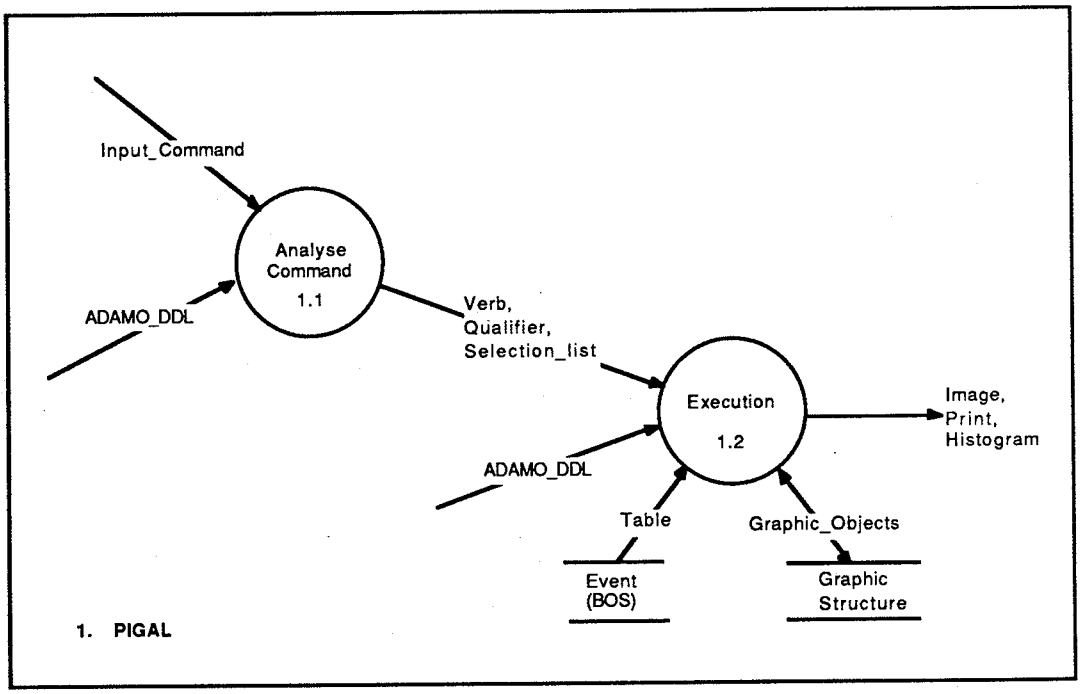
DDL : description formelle du diagramme précédent; la première section décrit les entités, la deuxième décrit les relations entre entités, la troisième est la définition des attributs.

```
DEFINE ESET
/*-------------------------------------------------------------------*/

ESDA    : 'Ecal storey data'
        = (Row,Column,Stack,Energy);

ECLU    : 'Ecal cluster'
        = (Charge,
           Energy(4),
           Radius(4),Theta(4),Phi(4),
           RadErr(4),ThErr(4),PhiErr(4),
           StackLimit(2) = INTE [1,2]
                         : 'Innermost and outermost stacks',
           ProcLevel     = INTE [0,*]
                         : 'Processing level');

END ESET

DEFINE RSET
/*-------------------------------------------------------------------*/

(ESDA [1,1] -> [0,*] EMOD): 'EcStorey data belong to a module';

(ESDA [0,1] -> [1,*] ECLU): 'Clusters are constructed from Storeys';

(ESDA [1,1] -> [0,1] EPAT): 'Neighbour pattern relation';

END RSET

DEFINE ATTRIBUTE
/*-------------------------------------------------------------------*/

Row         = INTE [1,228]: 'Storey row number';

Column      = INTE [1,384]: 'Storey column number';

Stack       = INTE [1,3]  : 'Stack number';

END ATTRIBUTE
```

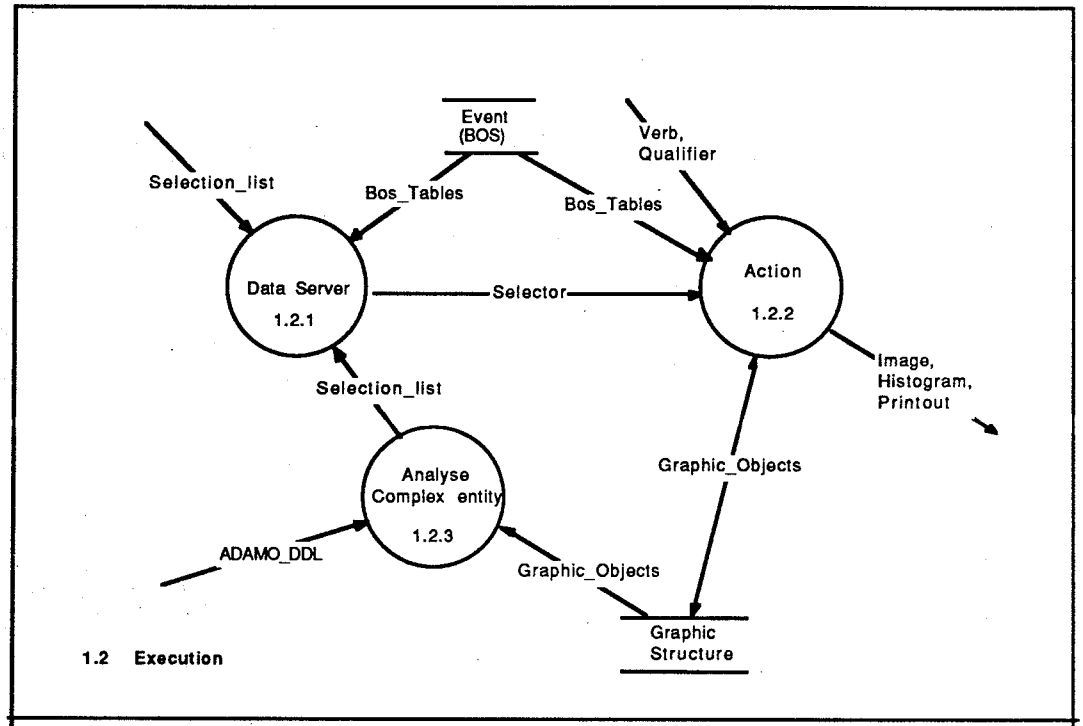
Annexe 2

Extrait de DFD de PIGAL

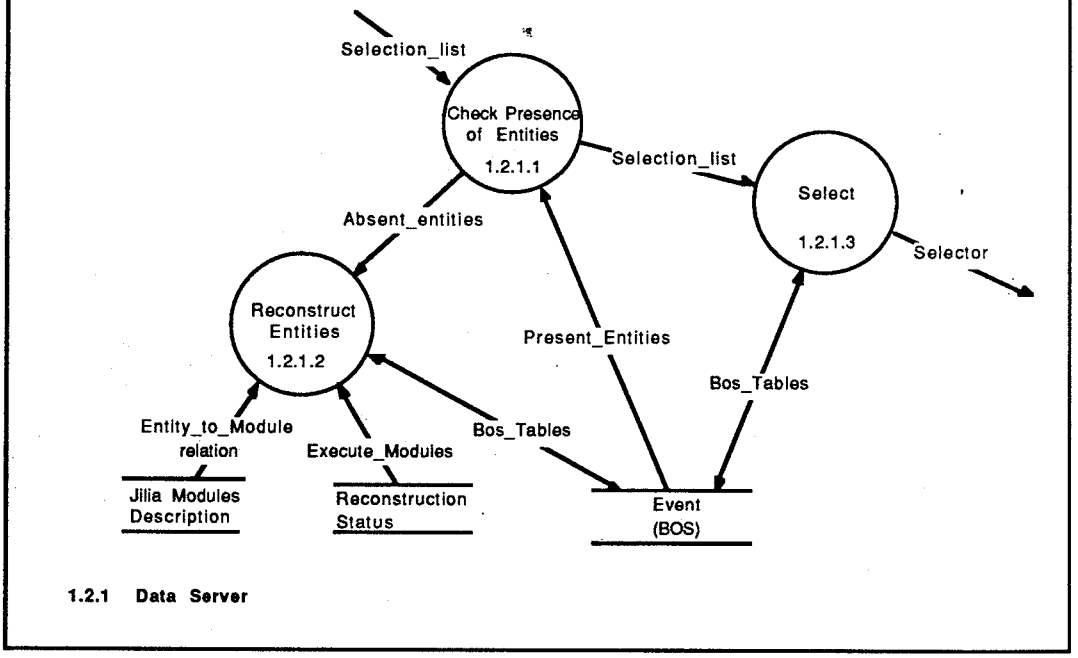
L'on trouvera ici un extrait de document DFD représentant la structure de PIGAL. A partir du premier DFD chaque processus peut être décrit par un nouveau DFD plus précis : 1.2, 1.2.1, 1.2.2 et ainsi de suite.



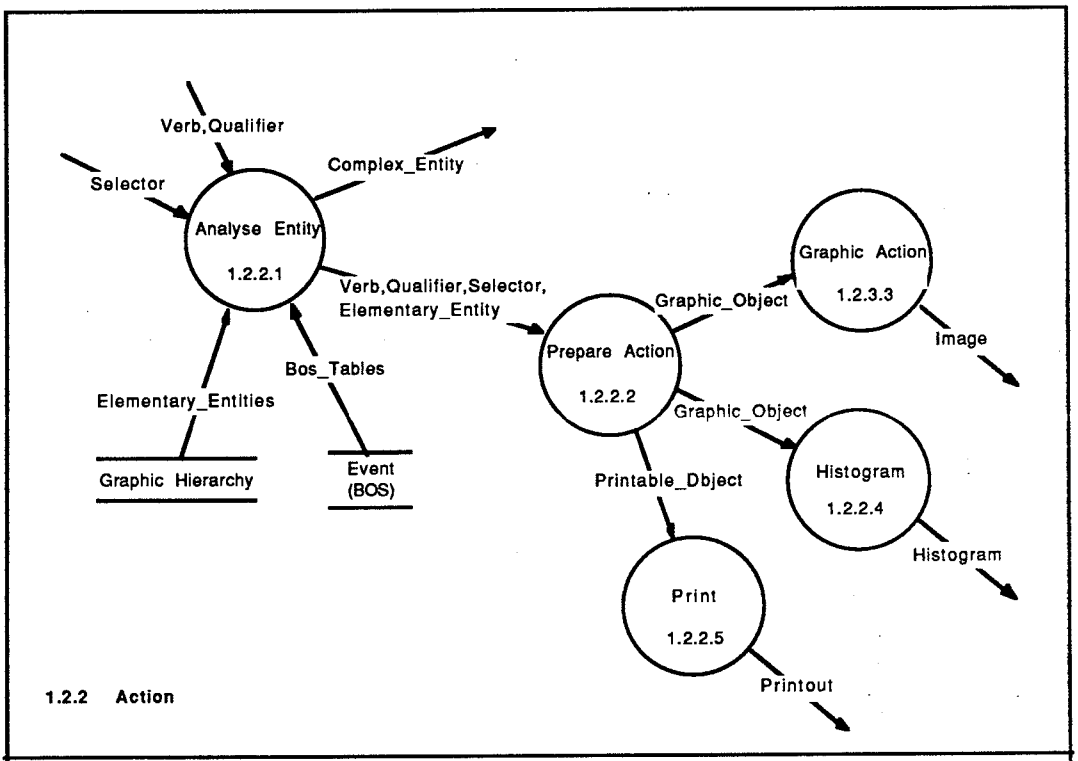
1. PIGAL

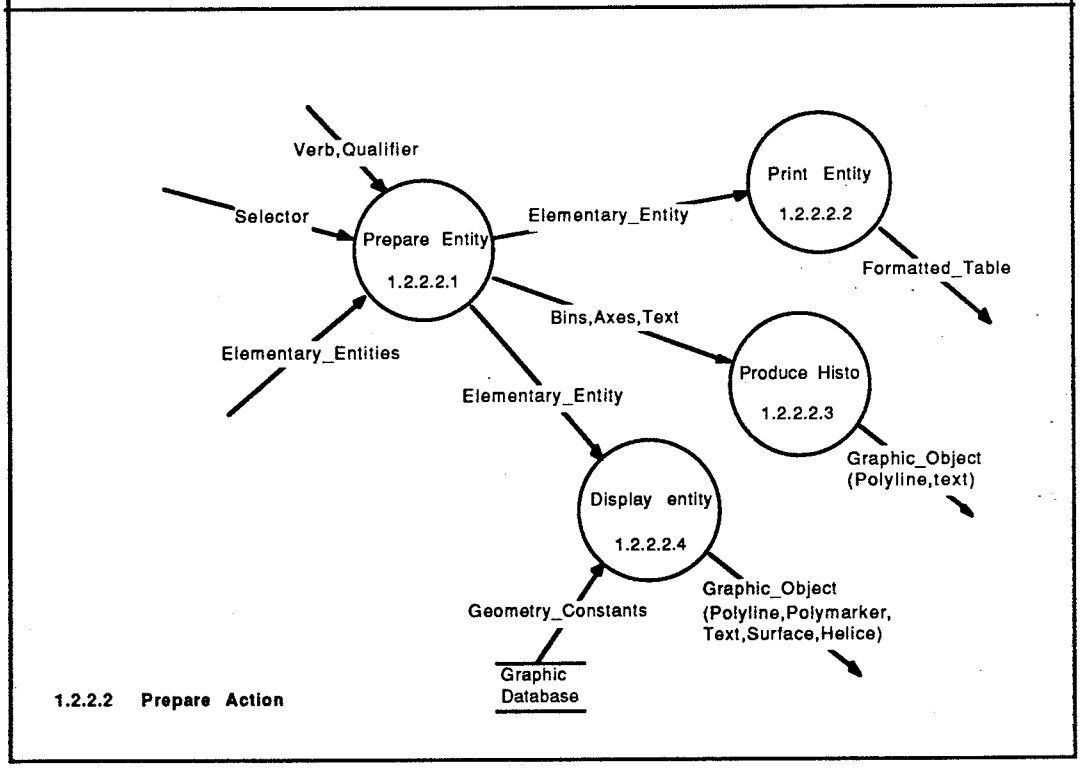


1.2 **Execution**



1.2.1 **Data Server**


Verb,Qualifier
Selector
Analyse Entity
1.2.2.1
Complex_Entity
Verb,Qualifier,Selector,
Elementary_Entity
Bos_Tables
Elementary_Entities
Graphic Hierarchy
Event
(BOS)
Prepare Action
1.2.2.2
Graphic_Object
Graphic Action
1.2.3.3
Image
Graphic_Object
Histogram
1.2.2.4
Histogram
Printable_Dbject
Print
1.2.2.5
Printout
1.2.2 Action


Verb,Qualifier
Selector
Prepare Entity
1.2.2.2.1
Elementary_Entity
Print Entity
1.2.2.2.2
Formatted_Table
Bins,Axes,Text
Elementary_Entities
Produce Histo
1.2.2.2.3
Elementary_Entity
Graphic_Object
(Polyline,text)
Display entity
1.2.2.2.4
Geometry_Constants
Graphic_Object
(Polyline,Polymarker,
Text,Surface,Helice)
Graphic
Database
1.2.2.2 Prepare Action

Annexe 3

Présentation simplifiée du langage PROLOG

PROLOG (PROgrammation en LOGique) [50] est un langage développé pour des applications d'intelligence artificielle. Il a été conçu et mis au point au GIA de Marseille (1972), et issu des travaux d'une équipe de recherche menée par A. Colmerauer à Marseille sur la compréhension du langage naturel [51] et d'une équipe menée par R.A.Kowalski à Londre sur la programmation en logique[52].

Il y a une grande différence entre PROLOG et les autres langages conventionnels. On dit souvent : " Un programme FORTRAN, PASCAL ... décrit un **ALGORITHME**, un programme PROLOG énonce des **FAITS** (axiomes, relations, proprietes)."

Le programme PROLOG est un ensemble de règles écrites par le programmeur. Chaque régle est formée d'une partie gauche (tête de règle) réduite à un seul terme, et d'une partie droite (queue de règle) formée d'une suite de n termes. Une flèche "->" sépare les deux parties. La règle termine par un ";". Par exemple

```
start ->initpigal
        initgks;
```

Si la queue d'une règle est vide, la règle s'appele "un fait", c'est à dire une formule toujours vraie, par exemple

```
verb(draw) ->;
```

signifie que "draw" est un verbe.

Les règles sont groupées en paquets commençant par le même identificateur, par exemple

```
verb(draw) ->;
verb(print) ->;
verb(histogram) ->;
```

Les termes peuvent contenir des variables qui sont locales à une règle, par exemple

```
chemin(a,b,c,d) -> shortpath(a,b,c,d);   (1)
chemin(a,b,c,d) -> longpath(a,b,c,d);    (2)
```

Ici **a,b,c,d** sont des variables, elles sont définies pour une seule règle. Donc les variables **a,b,c,d**, situées à gauche de la règle (1), désignent les mêmes choses que celles de droite, mais elles n'ont aucun rapport avec les variables de la règle (2).

Le mécanisme de PROLOG

Le mécanisme de PROLOG consiste en un effacement de buts représentés par des termes. Un but est "effacé" lorsque les contraintes lui correspondant sont vérifiées, par exemple

```
start ->initpigal
        initgks;
```

signifie que si l'initialisation de PIGAL est réussie et si l'initialisation de GKS est réussie alors le lancement de PIGAL est réussi.

Le fonctionnement de PROLOG est de chercher à effacer successivement les buts, c'est-à-dire pour l'exemple suivant :

```
go -> start initfor;          (1)
start -> initpigal;           (2)
initpigal ->;                 (3)
initfor ->;                   (4)
```

Soit **go** le but à effacer : PROLOG cherche à identifier (on dit **unifier**) **go** à la tête de règle 1, ce qui imposera d'effacer les nouveaux buts **start** et

initfor. Le terme **initfor** sera effacé immédiatement grace à la règle (4). Le terme **start** sera identifié par **initpigal** (règle 2), ainsi de suite jusqu'au moment tous les termes sont remplacés. Si un terme ne peut être remplacé, PROLOG retourne en arrière (remontée ou **backtracking**) et essaye un autre chemin pour l'effacer.

En général il y a plusieurs manières d'effacer un but, c'est-à-dire plusieurs réponses possibles, PROLOG les détermine toutes. Voyons un exemple trouvé dans PIGAL :

```
esda (thetaj,1,1) ->;          (1)
esda (energy,2,4) ->;          (2)
esda (eclu,3,9) ->;            (3)
```

On peut alors poser en PROLOG des questions sur cette base:
Quels attributs de l'entité ESDA sont les attributs réel ?

```
> esda(x,2,y);
```

PROLOG effectue les identifications suivautes :

- l'essai d'unification avec la règle (1) posera le système d'équations "x=thetaj, 2=1, y=1" : le système est insoluble, echec, PROLOG retourne en arrière.

- l'essai d'unification avec la règle (2) posera le système d'équations "x=energy, 2=2, y=4" : système est résolu avec les solutions affichées

```
x = energy,    y = 4
```

- l'essai d'unification avec la règle (3) posera le système d'équations "x=eclu, 2=3, y=9" : pas de solution.

Est-ce que ESDA a de relations avec d'autres entités ?

```
> esda(x,3,y);
```

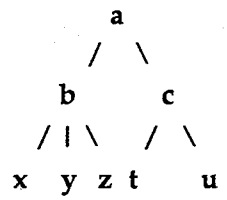
Les réponses sont :

```
x = eclu,   y = 9
```

Structures de données en PROLOG

PROLOG offre la possibilité de manipulation de variables ou de constantes. Les types de données sont très variées :

- nombres entiers ou réels
- chaînes de caractères
- identificateurs
- listes et n-uplets, ex.
  aa.bb.cc.nil est une liste, "**nil**" signifiant la fin de la liste
  <a,b,c> est un n-uplets
- arbres, ex.
  a (b (x, y, z), c (t, u)) représent une structure d'arbre



Récursivité

L'exemple suivant présente un type de règle utilisé beaucoup en PROLOG : la règle récursive.

```
extraire("") ->;
extraire(s) -> substring (s,1,4,u)
out(s)
conc-string(u,r,s)
extraire (r);
```

Le fonctionnement de cette règle est de tirer d'une chaîne de caractères les premiers 4 caractères chaque fois, de les mettre dans le variable **s** et de les imprimer. Le reste des caractères est gardé dans la variable **r** qui sera soumise à "extraire". La fonction se termine quand la chaîne devient vide.

Application de PROLOG

Une application typique de PROLOG comprend deux parties :

- une base de connaissances représentant les faits sur lesquels le système travaillera
- un programme qui manipule ces faits et répond aux questions posées par l'utilisateur

Dans certains systèmes les réponses peuvent être immédiatement intégrées comme nouvelle donnée dans la base de connaissances pour une utilisation ultérieure.

Interface avec les autres langages

PROLOG offre une interface avec les autres langages : FORTRAN, C ... Dans le cas de FORTRAN, les subroutines FORTRAN sont connues de PROLOG sous forme de prédicats externes, qui sont invoqués selon la syntaxe suivante:

```
subname(x,y, ...) -> /?300/?99;
```

où **subname** est le nom de prédicat; **x,y**... sont les arguments. Le signe **/?99** indique qu'il s'agit d'un prédicat externe. Le **300** sera transmis dans la subroutine **user_rule** appelée par l'interpréteur PROLOG et qui doit être fournie par l'utilisateur si des prédicats externes sont utilisés.

Dans **user_rule**, un aiguillage genre **goto** permet de faire le tri entre les diverses subroutines prévues. Chacune de ces subroutines doit effectuer le transfert des arguments entre PROLOG et FORTRAN en utilisant une série de services intégrés à la bibliothèque PROLOG: get_real, get_integer, get_string, put_real, put_integer, put_string. Les services get_term et put_term permettent de transmettre un terme, c'est à dire une variable structurée (arbre) en un seul appel.

Annexe 4

GKS et graphique

GKS (Graphic Kernel System) a été développé depuis 1976 et a évolué en un standard international. Il fournit un ensemble de fonctions permettant la réalisation de représentations graphiques sur différents types de machine. Il permet à ses applications d'être très facilement portable. En effet GKS constitue un noyau indépendant du système graphique de la machine sur laquelle il est implémenté.

Il est nécessaire de présenter de manière générale certains concepts élémentaires de GKS utilisés dans PIGAL

Les sorties sur écran :
La tâche de GKS est d'engendrer des dessins. Ceux-ci sont composés de primitives (lignes, cercles, cubes ...). Les primitives les plus utilisées sont : ligne (polyline), marqueur (polymarker), chaîne de caractères (text) et surface (fill area).

Les systèmes de coordonnées et les transformations :
Les primitives de sortie sont créées dans un ou plusieurs systèmes de coordonnées. Un ensemble de transformations définissent le passage entres différents systèmes de coordonnées.

Les stations de travail :
Les appareils graphiques sont divisés en groupes appelés Workstation. Celles-ci pouvant désigner un écran, une table traçante, etc.

Les vues :
Une vue peut contenir une partie ou un dessin complet. Chaque vue peut être transformée indépendamment les unes des autres. De plus une

Workstation peut contenir plusieurs vues.

Les segments :
Un dessin complexe est composé d'un ensemble de segments (lignes, cercles ...) pouvant être détruits ou cachés indépendamment les uns des autres. Lorsqu'on dessine un objet amené à rester sur l'écran, on doit auparavant ouvrir un segment et le fermer à la fin du dessin de l'objet. Ainsi, tous les points dessinés entre l'ouverture et la fermeture du segment possèderont un même identificateur. Un segment est unique dans l'écran.

L'identificateur de "pick" :
Il s'agit d'un numéro arbitraire associé à un segment de l'écran. On peut spécifier ce paramètre avant un dessin, il restera jusqu'au prochain recouvrement (spécification d'un autre identificateur de pick). Il est indépendant du numéro de segment. La différence essentielle entre ces deux paramètres vient du fait que l'identificateur de pick ne représente pas une zone connexe dans l'écran. Il est possible que les différents numéros de segment soient associés à un même identificateur de pick. L'identificateur de pick ainsi que le numéro de segment peuvent être récupérés par un "clic" sur l'écran, identifiant ainsi complètement l'objet indiqué.

Le méta-fichier :
C'est un moyen de stocker les images dans un but de création d'archive ou de transfert d'images (en une position différente de l'écran ou dans une autre station).

Annexe 5

Interface Client/Serveur_Trigger

L'on trouvera ici deux "Stubs" de l'interface Client/Serveur_Trigger, créés automatiquement par le compilateur RPC pour le fichier de définition donné Fig.32 (page 90). La première section est assemblée avec Serveur_Trigger, la seconde est liée avec Client. Ces deux "Stubs" assurent la communication entre deux processus à travers la bibliothèque d'exécution de RPC.

```
C       SERVER STUB routines for package RSUBT
C       ======================
C
C       Generated automatically by the RPC Compiler
C
        SUBROUTINE R_TRIGGER(RPC_P_BUF)
        INTEGER RPC_P_BUF,RPC_A,RPC_B
        CHARACTER*80 C
        CALL UPK_VSTRING_FOR(RPC_P_BUF,C,80)
        CALL RPC_INIT_RETURN_FOR(RPC_P_BUF)
        CALL RPC_EARLY_RETURN(RPC_P_BUF)
        CALL TRIGGER(C)
        END

        SUBROUTINE R_TRIG_DPL_INIT(RPC_P_BUF)
        INTEGER RPC_P_BUF
        CALL RPC_INIT_RETURN_FOR(RPC_P_BUF)
        CALL TRIG_DPL_INIT
        END

        SUBROUTINE R_T56(RPC_P_BUF)
        INTEGER RPC_P_BUF,RPC_A,RPC_B
        CHARACTER*80 C
        CALL UPK_VSTRING_FOR(RPC_P_BUF,C,80)
        CALL RPC_INIT_RETURN_FOR(RPC_P_BUF)
        CALL T56(C)
        END

C
C                       Main stub entry point
C
```

```
      SUBROUTINE R_RSUBT(RPC_P_BUF)
      INTEGER RPC_P_BUF
      INTEGER RPC_REQUEST, STATUS
      PARAMETER(RPC_S_UNSUPPORTED_VERSION=139624458)
      PARAMETER(RPC_S_BAD_PROCEDURE_NUMBER=139624466)

      CALL UPK_SHORT_FOR(RPC_P_BUF, RPC_REQUEST)
      IF ((RPC_REQUEST.NE.0).AND.(RPC_REQUEST.NE.4150)) THEN
      CALL RPC_SET_ERROR(RPC_P_BUF,RPC_S_UNSUPPORTED_VERSION)
      ELSE
         CALL UPK_SHORT_FOR(RPC_P_BUF, RPC_REQUEST)
         IF ((RPC_REQUEST.LE.0).OR.(RPC_REQUEST.GT.3)) THEN
            CALL RPC_SET_ERROR(RPC_P_BUF,RPC_S_BAD_PROCEDURE_NUMBER)
         ELSE
            GOTO(10,20,30),RPC_REQUEST

   10       CALL R_TRIGGER(RPC_P_BUF)
            GOTO 888

   20       CALL R_TRIG_DPL_INIT(RPC_P_BUF)
            GOTO 888

   30       CALL R_T56(RPC_P_BUF)
            GOTO 888
  888       CONTINUE

         END IF
      END IF
      END

C     Call this procedure at initialisation time ***
      SUBROUTINE ATTACH_RSUBT
      EXTERNAL R_RSUBT
      INTEGER STATUS, PROG_NO

      CALL RPC_ATTACH_STUB_FOR(STATUS,R_RSUBT,
     +'RSUBT                                    ',PROG_NO)
      CALL RPC_REPORT_ERROR(STATUS)
      END


C-----------------------------------------------------------------------


C     CLIENT STUB routines for package RSUBT
C     ======================
C
C     Generated automatically by the RPC Compiler
C
      SUBROUTINE TRIGGER(C)
      COMMON /C_RSUBT/ H_RSUBT
      INTEGER H_RSUBT
      CHARACTER*(*) C
```

```
      INTEGER RPC_P_BUF,RPC_A,RPC_B
      CALL RPC_BEGIN_CALL_FOR(RPC_P_BUF, H_RSUBT,82,0,4150,1)
      CALL PCK_STRING_FOR(RPC_P_BUF,C)
      CALL RPC_CALL(H_RSUBT,RPC_P_BUF,400)
      CALL RPC_END_CALL_FOR(RPC_P_BUF)
      RETURN
      END

      SUBROUTINE TRIG_DPL_INIT
      COMMON /C_RSUBT/ H_RSUBT
      INTEGER H_RSUBT
      INTEGER RPC_P_BUF
      CALL RPC_BEGIN_CALL_FOR(RPC_P_BUF, H_RSUBT,0,0,4150,2)
      CALL RPC_CALL(H_RSUBT,RPC_P_BUF,-1)
      CALL RPC_END_CALL_FOR(RPC_P_BUF)
      RETURN
      END

      SUBROUTINE T56(C)
      COMMON /C_RSUBT/ H_RSUBT
      INTEGER H_RSUBT
      CHARACTER*(*) C
      INTEGER RPC_P_BUF,RPC_A,RPC_B
      CALL RPC_BEGIN_CALL_FOR(RPC_P_BUF, H_RSUBT,82,0,4150,3)
      CALL PCK_STRING_FOR(RPC_P_BUF,C)
      CALL RPC_CALL(H_RSUBT,RPC_P_BUF,-1)
      CALL RPC_END_CALL_FOR(RPC_P_BUF)
      RETURN
      END

C     Call this procedure at initialisation time ***
      SUBROUTINE OPEN_RSUBT
      COMMON /C_RSUBT/ H_RSUBT
      INTEGER H_RSUBT
      INTEGER STATUS
      CALL RPC_OPEN_FOR(STATUS,H_RSUBT,
     +'RSUBT                                    ')
      CALL RPC_REPORT_ERROR(STATUS)
      END
```

Annexe 6

Mécanisme de transfert de message

Le mécanisme de transfert de message entre Client et Serveur, on prend Serveur_JULIA comme exemple.

Partie 1 (les subroutines s'exécutant au côté **Client**)

INIRPCJ : créer Serveur_JULIA et l'initialiser

USER_RULE : interface PROLOG-FORTRAN

PROUSE : subroutine pour le branchement d'appels venant de PROLOG, notamment pour l'appel CALLSR

CALLSR : former les messages et les envoyer au Serveur concerné par l'appel RPC spécifique pour chaque Serveur

MOVPOI,INARG,ADDST,ADDINADDRE,GETREA,GETINT, GETSTR : subroutines pour traiter d'arguments

Partie 2 (les subroutines s'executant au côté **Serveur_JULIA**)

CALLSUJ : appelée par le systeme d'exécution de RPC. Il récupère le nom du service demandé et la liste d'arguments, fait appel à ce service

PSBADD,LSUBR : pour trouver l'adresse du service

ICONV,PRLIST : préparer la liste d'arguments codée selon les conventions du système d'exploitation (VMS)

INISUBJ : la liste des services fournis par Serveur_JULIA et connus par le Client

Partie 1 :

```
      SUBROUTINE INIRPCJ
C
C  create julia process and initialize julia
           character*80 title
           call open_rsubj
           call attach_lsubj
           title = '           JULIA   Server   Window'
           call jterminal(title)
           return
           end
C
      SUBROUTINE USER_RULE(NB,ERROR_FOUND,ERROR_NB)
C
C   Interface PROLOG-FORTRAN
C
      CALL PROUSE (NB,ERROR_FOUND,ERROR_NB)
      RETURN
      END
C
      SUBROUTINE PROUSE (NB,ERROR_FOUND,ERROR_NB)
C
C   Interface prolog program
C    A number is associated to each fortran module
C     which can be called with a computed GO TO
C
C-------------------------------------------------------------
      INTEGER NB,ERROR_NB
      LOGICAL ERROR_FOUND
      CHARACTER*132 STRING,STRINGI,STRINGO,STRINGP
      ERROR_FOUND=.FALSE.
      IND=(NB-500)/10
      GO TO
     * (999,520,530,540,550,560,570,580,590,600)IND
  520 CALL INARG
      GO TO 999
  530 CALL GET_STRING(1,L,%ref(STRING),ERROR_FOUND)
      CALL GET_INTEGER(2,LRES,ERROR_FOUND)
      CALL CLTOU(STRING(1:L))
      CALL ADDST(STRING(1:L),LRES)
      GO TO 999
  540 CALL GET_INTEGER(1,IVAL,ERROR_FOUND)
      CALL ADDIN(IVAL)
      GO TO 999
  550 CALL GET_REAL(1,VAL,ERROR_FOUND)
      CALL ADDRE(VAL)
      GO TO 999
  560 CALL GET_STRING(1,L,%ref(STRING),ERROR_FOUND)
      CALL GET_STRING(2,LI,%ref(STRINGI),ERROR_FOUND)
      CALL CLTOU(STRING(1:L))
```

```
      CALL CLTOU(STRINGI(1:LI))
      CALL CALLSR(STRING(1:L),STRINGI(1:LI))
      GO TO 999
  570 CALL GETREA(VALUE)
         CALL PUT_REAL(1,VALUE,ERROR_FOUND)
      GO TO 999
  580 CALL GETINT(IVAL)
      CALL PUT_INTEGER(1,IVAL,ERROR_FOUND)
      GO TO 999
  590 CALL GETSTR(STRINGO,LO)
      CALL PUT_STRING (1,LO,%ref(STRINGO),ERROR_FOUND)
      GO TO 999
  600 CALL MOVPOI
      GO TO 999
  999 CONTINUE
      RETURN
      END
C
      SUBROUTINE CALLSR(NAMEI,PROCI)
C
C   send message to Server
C
      LOGICAL PAW_FIRST/.TRUE./
      LOGICAL HISTO_FIRST/.TRUE./
      CHARACTER*(*) NAMEI,PROCI
      CHARACTER*500 MEMOR,N2
      DIMENSION IDESCR(2,50),N3(2,50)
      COMMON /ARGLIS/NARG,MEMOR,IDESCR,IPOS
C
C     ACTIVATION DE CE PROCESS
C
        LNAM=LEN(NAMEI)
        MEMOR(IPOS+1:IPOS+LNAM)=NAMEI
        IDESCR(1,NARG+1)=IPOS+1
        IDESCR(2,NARG+1)=LNAM+3
        IPROC=1
C
C  rpc call
C  argument from common block to rpc
C
      CALL UCOPY(IDESCR(1,1),N3(1,1),100)
      N2(1:500) = MEMOR(1:500)
      N1 = NARG
      N4 = IPOS
C
C  rpc calls for different Servers
C
      IF(PROCI(1:6).EQ.'DIALOG') CALL CALLSUD(N1,N2,N3,N4)
      IF(PROCI(1:5).EQ.'PIGAL') CALL CALLSUG(N1,N2,N3,N4)
      IF(PROCI(1:5).EQ.'JULIA'.OR.PROCI(1:6).EQ.'SELTAB')
     +                          CALL CALLSUJ(N1,N2,N3,N4)
      IF(PROCI(1:5).EQ.'HISTO') CALL CALLSUH(N1,N2,N3,N4)
      IF(PROCI(1:3) .EQ. 'PAW') THEN
```

```
            IF(PAW_FIRST) THEN
                 CALL INIRPCP
                 PAW_FIRST = .FALSE.
                 RETURN
            ENDIF
            CALL PPAW
       ENDIF
C
C argument from rpc to common block
C
       CALL UCOPY(N3(1,1),IDESCR(1,1),100)
       MEMOR(1:500) = N2(1:500)
       NARG = N1
       IPOS = N4
       RETURN
       END
C
       SUBROUTINE MOVPOI
C
C   MOVE THE ARGUMENTS POINTER WITHOUT DOING ANYTHING ELSE
C
       CHARACTER*500 MEMOR
       DIMENSION IDESCR(2,50)
       COMMON /ARGLIS/NARG,MEMOR,IDESCR,IPOS
       NARG=NARG+1
       RETURN
       END
C
       SUBROUTINE INARG
C
C  Initialises the arguments list,
C  which has to be filled before each subroutine call
C
       CHARACTER*500 MEMOR
       DIMENSION IDESCR(2,50)
       COMMON /ARGLIS/NARG,MEMOR,IDESCR,IPOS
       NARG=0
       IPOS=0
       RETURN
       END
C
       SUBROUTINE ADDST(STRING,LRES)
C
C  Adds a string into the arguments list.
C  A descriptor is built, and (on vax) its address is inserted
C  into the list
C
       CHARACTER*500 MEMOR
       DIMENSION IDESCR(2,50)
       COMMON /ARGLIS/NARG,MEMOR,IDESCR,IPOS
       CHARACTER*(*)STRING
       NARG=NARG+1
       LSTR=LEN(STRING)
```

```
      LR=LRES
      IF(LR.LT.LSTR)THEN
         LR=LSTR
      ENDIF
      NORG=IPOS+1
      IPOS=IPOS+LR
      NEND=NORG+LSTR-1
      MEMOR(NORG:NEND)=STRING
      DO 10 IC=NEND+1,NCHAR
   10 MEMOR(IC:IC)=' '
      IDESCR(1,NARG)=NORG
      IDESCR(2,NARG)=LR+3
      RETURN
      END
C
      SUBROUTINE ADDIN(IVAL)
C
C  Adds integer to the arguments list
C   On vax, insert its address in the args list;
C   On IBM, fill the descriptor for it
C
      CHARACTER*500 MEMOR
      DIMENSION IDESCR(2,50)
      COMMON /ARGLIS/NARG,MEMOR,IDESCR,IPOS
      NARG=NARG+1
      IDESCR(1,NARG)=IVAL
      IDESCR(2,NARG)=1
      RETURN
      END
C
      SUBROUTINE GETREA(VAL)
C
C  Get next real number from the args list
C
      CHARACTER*500 MEMOR
      DIMENSION IDESCR(2,50)
      COMMON /ARGLIS/NARG,MEMOR,IDESCR,IPOS
      EQUIVALENCE(RLOC,ILOC)
      NARG=NARG+1
      ILOC=IDESCR(1,NARG)
      VAL=RLOC
      RETURN
      END
C
      SUBROUTINE GETINT(IVAL)
C
C   Get next integer number from the args list
C
      CHARACTER*500 MEMOR
      DIMENSION IDESCR(2,50)
      COMMON /ARGLIS/NARG,MEMOR,IDESCR,IPOS
      NARG=NARG+1
      IVAL=IDESCR(1,NARG)
```

```
      RETURN
      END
C
      SUBROUTINE GETSTR(STRING,L)
C
C   GET A CHARACTER STRING FROM THE ARGUMENTS LIST.
C     ON VAX : TRANSMIT THE DESCRIPTOR TO DECONV ROUTINE
C    ON IBM : EXTRACT THE ADDRESS IN THE GLOBAL STRING (CHARS), AND
C       LENGTH FROM THE DESCRIPTOR,
C    THEN, EXTRACT CHARACTERS. THERE IS A CHECK ON VALUES OF
C      BEGIN AND END
C
      CHARACTER*500 MEMOR
      DIMENSION IDESCR(2,50)
      COMMON /ARGLIS/NARG,MEMOR,IDESCR,IPOS
      CHARACTER*(*) STRING
      NARG=NARG+1
      NDEB=IDESCR(1,NARG)
      L=IDESCR(2,NARG)-3
      NEND=NDEB+L-1
      STRING(1:L)=MEMOR(NDEB:NEND)
      RETURN
      END
C
      SUBROUTINE DECONV(STRINGI,STRINGO,L)
C
C   RETURN A STRING, GIVEN A STRING.
C   NECESSARY FOR GETSTR ON VAX.
C   NOT USED ON IBM
C
      CHARACTER*(*) STRINGI,STRINGO
      L=LEN(STRINGI)
      STRINGO(1:L)=STRINGI
      RETURN
      END
C
      SUBROUTINE ADDRE(VAL)
C
C Adds real into the arguments list
C   On vax, insert its address in the args list;
C   On IBM, fill the descriptor for it
C
      CHARACTER*500 MEMOR
      DIMENSION IDESCR(2,50)
      COMMON /ARGLIS/NARG,MEMOR,IDESCR,IPOS
      EQUIVALENCE (ILOC,RLOC)
      NARG=NARG+1
      RLOC=VAL
      IDESCR(1,NARG)=ILOC
      IDESCR(2,NARG)=2
      RETURN
      END
C
```

Partie 2 :

```
      SUBROUTINE CALLSUJ(N1,N2,N3,N4)
C
C  calls any subroutine, referred by its name NAME
C  On vax, the table ILST contains the addresses of the arguments
C  list, which is the arguments list for the fortran call.
C  This is passed to lib$callg, which will make the call
C
      CHARACTER*500 MEMOR,N2
      DIMENSION IDESCR(2,50),N3(2,50)
      COMMON /ARGLIS/ NARG,MEMOR,IDESCR,IPOS
      CHARACTER*32 NAML
      CHARACTER*132 AL
      PARAMETER (MXSUBR=200)
      CHARACTER*32 NAMES(MXSUBR)
      COMMON/SUBROU/NAMES,IADSUB(MXSUBR),NSUBR
      INTEGER ILST(-1:50),IDSC(2,50)
      DATA ILST/ZC2E90000,51*0./
C
C argument from rpc to common block
C
      CALL UCOPY(N3(1,1),IDESCR(1,1),100)
      MEMOR(1:500) = N2(1:500)
      NARG = N1
      IPOS = N4
      NAML(1:32)=' '
      IST=IDESCR(1,NARG+1)
      IEND=IDESCR(2,NARG+1)+IST-4
      NAML=MEMOR(IST:IEND)
      write(6,*)'  callsu :namL,IST,IEND=',namL,IST,IEND
      CALL PRLIST(ILST,IDSC)
      DO 10 IS=1,NSUBR
      IF(NAML.EQ.NAMES(IS))THEN
         IF(IADSUB(IS).EQ.-1)GO TO 41
C
C  make the call : ILST is the arguments list; IADSUB(IS) contains the
C  address of the entry point of the subroutine to be called
C
         CALL LIB$CALLG(ILST(0),%VAL(IADSUB(IS)))
         NARG=0
         NNUMB=0
         GO TO 999
      ENDIF
   10 CONTINUE
      WRITE(6,*)' Gee, I do not know this ',namL
      WRITE(6,*)' will be skipped'
  999 CONTINUE
C
C argument from common block to rpc
C
```

```
      CALL UCOPY(IDESCR(1,1),N3(1,1),100)
      N2(1:500) = MEMOR(1:500)
      N1 = NARG
      N4 = IPOS
      RETURN
      END
C
      SUBROUTINE PSBADD(NAME,SUBRL)
C
C  get the address of one service
C
      CHARACTER*(*) NAME
      PARAMETER (MXSUBR=200)
      CHARACTER*32 NAMES(MXSUBR)
      COMMON/SUBROU/NAMES,IADSUB(MXSUBR),NSUBR
      DATA N/0/
      NSUBR=NSUBR+1
      NAMES(NSUBR)='                                                '
      NAMES(NSUBR)=NAME
      IADSUB(NSUBR)=LSUBR(SUBRL)
      RETURN
      END
C
      FUNCTION LSUBR(IX)
C
C   FIND THE ADDRESS OF AN EXTERNAL SUBROUTINE, AND RETURN IT
C
      LSUBR=%LOC(IX)
      RETURN
      END
C
      SUBROUTINE ICONV(IA,IB)
C
C   Puts a string descriptor (in IA) into a variable IB
C
      DIMENSION IA(2),IB(2)
      IB(1)=IA(1)
      IB(2)=IA(2)
      RETURN
      END
C
C
      SUBROUTINE PRLIST(ILST,IDSC)
C
C   Prepare arguments list for subroutine call
C
      CHARACTER*500 MEMOR
      DIMENSION IDESCR(2,50)
      COMMON /ARGLIS/ NARG,MEMOR,IDESCR,IPOS
      PARAMETER (MXSUBR=200)
      CHARACTER*32 NAMES(MXSUBR)
      COMMON/SUBROU/NAMES,IADSUB(MXSUBR),NSUBR
      INTEGER ILST(-1:50),IDSC(2,*)
```

```
      ILST(0)=NARG
      DO 10 I=1,NARG
      IF(IDESCR(2,I).GT.3)THEN
         NDEB=IDESCR(1,I)
         L=IDESCR(2,I)-3
         NEND=NDEB+L-1
         CALL ICONV(MEMOR(NDEB:NEND),IDSC(1,I))
         ILST(I)=%LOC(IDSC(1,I))
      ELSE
         ILST(I)=%LOC(IDESCR(1,I))
      ENDIF
   10 CONTINUE
      RETURN
      END
C
      SUBROUTINE INISUBJ
C
C   Initialises the addresses and names lists of foreseen
C   subroutines (service) to be called
C
      PARAMETER (MXSUBR=200)
      CHARACTER*32 NAMES(MXSUBR)
      COMMON/SUBROU/NAMES,IADSUB(MXSUBR),NSUBR
      EXTERNAL SELTAB
      EXTERNAL SELIN
           .
           .
      EXTERNAL CRLCEB
      EXTERNAL XOUTBK
      EXTERNAL XIMPORT
      CALL PSBADD('SELTAB',SELTAB)
      CALL PSBADD('SELIN',SELIN)
           .
           .
      CALL PSBADD('CRLCEB',CRLCEB)
      CALL PSBADD('XOUTBK',XOUTBK)
      CALL PSBADD('XIMPORT',XIMPORT)
      RETURN
      END
```

Bibliographie

1 LEP - DESIGN; **Report Cern-Lep**, 84-01-Vol II

2 Physics at LEP; Cern 86-02 (1986)

3 Experiments at CERN in 1989

4 ALEPH Collaboration, A precise determination of the number of families with light neutrinos and of the Z boson partial widths; **Phys. Lett. B** Volume 235, number 3,4

5 ALEPH Collaboration, Determination of the leptonic branching ratios of the Z; **Phys. Lett. B** Volume 234, number 3

6 ALEPH Collaboration, Heavy Flavour Production in Z Decays; CERN-EP/90-54, 24 April 1990 (Submitted to Physique Letters B)

7 ALEPH Collaboration, Search for the Neutral Higgs Boson from Z^o Decay in the Higgs Mass Range between 11 and 24 GeV; **Phys. Lett. B** Volume 241, number 1

8 J.V.Allaby, Data acquisition and analysis at LEP; Proceedings of the 1987 CERN School of Computing, **CERN Yellow Report**, CERN 88-03, p.240

9 The ALEPH Collaboration, ALEPH: A Detector for Electron-Positron Annihilations at LEP; **CERN-EP/90-25**, 22 February 1990 (submitted to Nucl. Inst. Meth.)

10 M.Delfino et al., The ALEPH event reconstruction facility: parallel processing using workstation; **Proceeding of the International Conference on Computing in High Energy Physics**, Oxford England, 10-14 April 1989, p.401

11 T.DeMarco, Structured Analysis and System Specification; **Yourdon** (1978)

12 G.Kellner, Software engineering; Proceedings of the 1988 CERN School of Computing, **CERN Yellow Report**, CERN 89-06, p.8

13 S.M.Fisher, The practice of SA-SD; Proceedings of the 1988 CERN School of Computing, **CERN Yellow Report**, CERN 89-06, p.34

14 P.P.Chen, the Entity-Relationship Model - Toward a Unified View of Data; **ACM Transaction on Database Système**, Vol.1, No 1, March 1976, pp 9-36

15 Z.Qian et al., Use of the ADAMO data management system within ALEPH; **Computer Phys. Commun.** 45 (1987) p.283

16 M.G.Green, The ADAMO Data System, an Introduction for Particle Physicists; **Royal Holooway and Bedford New College** 89-01 (1989)

17 S.M.Fisher and P.Palazzi, Using a Data Model from software design to data analysis: what have we learned? **Proceeding of the International Conference on Computing in High Energy Physics**, Oxford England, 10-14 April 1989, p.169

18 R.Johnson, Particle identification for dE/dx in Aleph; **Aleph Note** 88-51 (1988)

19 A.Ealet, Application de l'analyse multidimensionnelle à la reconnaissance de photons dans le calorimètre d'Aleph; thèse d'état

20 J.Knoblock et P.Norton, status of reconstruction algorithms for Aleph; **Aleph Note** 88-6 (1988)

21 J.P.Albanese et al., **Aleph Note** 84-33 (1984), 85-153 (1985), 86-58 (1986)

22 V.Blobel, **DESY Internal Report**; DESY R1-88-01 (January 1988)

23 H.Albrecht et al., ALPHA User's Guide; **ALEPH** 89-151, **Softwr** 89-22, 19 September 1989

24 R.Brun et al., PAW - Physics Analysis Workstation; **CERN program library** Q121, August 1989

25 T.H.Burnett, IDA: an interactive data analysis environment for high energy physics; **Computer Phys. Commun.** 45 (1987)

26 A.Aimar et al., Table Interaction and Plotting; TIP user guide, version 1.0, 19 June 1989

27 C.Grab, DALI - Event Display User's Guide; **ALEPH Note** 89-12, 22.11.1989

28 A.Bonissent and F.Etienne, Artificial intelligence steering for the interactive analysis of a high energy physics experiment; **Proceeding of the International Conference on Computing in High Energy Physics**, Oxford, England, 10-14 April 1989

29 A.Bonissent and F.Etienne, PIGAL - Prolog interactive and graphics for Aleph analysis; Pigal user's guide, vers 2.24, June 1989

30 S.M.Fisher, EARL: Entity Relationship query language; **ALEPH SOFTWR** 88-5 2.5.1988

31 R.Brun and D.Lienart, HBOOK User Guide; **CERN Computer Centre Program Library, Long Write-up**, October 28, 1987

32 A.Bonissent et al., Loosely Coupled Distributed Architecture for Interactive Event Display and Analysis; **Proceedings of the International workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics**; Lyon France, March 1990

33 H.E.Bal et al., Programming Languages for Distributed Computing Systems; **ACM Computing Surveys**, Vol. 21, No. 3, September 1989

34 B.Liskov, Structure of Distributed Programs; Invited talk of 12th Internationnal Conference on Software Engineering, Nice France, March 26-30 1990

35 S.Mullender et al., Distributed Systems; **ACM Press** Frontier Series

36 B.J.Nelson, Remote Procedure Call; **XEROX PARC** CSL-81-9, May 1981

37 A.Birrel and B.Nelson, Implementing Remote Procedure Calls; **ACM Transactions on Computer System**, Vol.2, No.1, Jan 1984

38 T.J.Berners-Lee, RPC User Manual; Version 2.4.0, Last revised July 1989

39 T.J.Berners-Lee, RPC Internals; Version 2.3.1, Last revised February 1989

40 R.Brun and J.Zoll, ZEBRA - Data Structure Management System; **CERN Program Library** Q100, 1989

41 Digital Equipment Corporation, Guide to VAX Performance and Coverage Analyzer; April 1989

42 Allocating Modules to processors in a Distributed System; **IEEE Trans. Software Engineering**, November 1989

43 W. W.Chu et al., Task allocation in distributed data processing; **Computer**, pp.57, November 1980

44 R.Hayes and R.D.Schlichting, Facilitating Mixed Language Programming in Distributed System; **IEEE Trans. Software Engineering**, December 1987

45 B.N.Bershad et al., A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems; **IEEE Trans. Software Engineering**, August 1987

46 J.C.Wileden et al., Specification Level Interoperability; **Proceeding of 12th International Conference on Software Engineering**, Nice France, 26-30 March 1990

47 R.W.Scheifler and J.Gettys, The X Window System; **ACM Transaction on Graphic**, Vol.5, No.2, April 1986

48 O.Mathieu, Etude de l'utilisation des méthode de l'intelligence artificielle en analyse physique; thèse d'état

49 D.E.Hall et al., The software bus: a vision for scientific software development; **Proceeding of the International Conference on Computing in High Energy Physics**, Oxford, England, 10-14 April 1989

50 F.Giannesini et al., PROLOG; **InterEditions**, Paris, 1985

51 A.Colmerauer et al., PROLOG, bases théoriques et développements actuels; **Techniques et sciences informatiques**, numéro 4, 1983

52 W.F.Clocksin and C.S.Mellish, Programming in PROLOG; **Springler-Verlag** 1981

53 W.Atwood et al., The Reason Project; **SLAC-PUB**-5242, April 1990

54 G.Pujoll and M.Schwartz, Réseaux locaux; Informatiques, **Edition Eyrolles** 1988

Glossaire

ADAMO	(Aleph DAta MOdel) Un logiciel développé dans ALEPH pour organiser les données selon le modèle Entité-Association
ALEPH	L'un de quatre expériences de physique des particules au LEP
ALEPHLIB	Bibliothèque des utilitaires d'ALEPH
ALPHA	Un logiciel d'analyse physique d'ALEPH
Appel séquentiel	(Sequentiel Call) Dans un logiciel, Client et Serveur n'exécutent pas en même temps
Appel concurrent	(Concurrent Call) Dans un logiciel, Client et Serveur exécutent en même temps
BOS	Un logiciel de gestionnaire de mémoire utilisé dans ALEPH
Callback	Un mécanisme de communication entre processus, la procédure appelée fait appel à une subroutine située dans le programme appelant
CERN	Organisation Européenne pour la Recherche Nucléaire
Client/Serveur	Un modèle de communication utilisé largement dans des applications réparties, choisi par PIGAL
Consommateur	Consommateur et Producteur sont les deux termes par rapport aux données partagées. Producteur est un programme qui exporte les données, tandis que Consommateur les importe.

DALI	Un système interactif graphique 2D utilisé dans ALEPH
DDL	(Data Definition Language) Langage formel de définition de données pour décrire en détail les structure de données
DEC	Digital Equipment Corporation
DFD	(Data Flow Diagram) La représentation graphique de SA (Structured Analysis)
EARL	(Entity and Relationship query Language) Un langage quasi naturel conçu pour exprimer les opérations dans la sélection des données ayant une structure ER
ECAL	(Electromagnetic CALorimeter) Le Calorimètre électromagnétique, un type de détecteur de particules
Echange de message	Voir Passage de message
ECLU	(Ecal cluster) Nom de l'entité de clusters du calorimètre électromagnétique
Entité-Association	(Entity-Relationship) Une stratégie pour définir la structure des données, utilisée dans ALEPH
ER	(Entity-Relationship) Voir Entité-Association
ERD	(Entity-Relationship Diagram) Le diagramme représentant le modèle Entité-Association
ESDA	(Ecal storey data) Nom de l'entité de cellules du calorimètre électromagnétique
FALCON	(Facility for ALeph COmputing and Networking) Ensemble de matériels (station de travail, réseau, disque), basé sur une architecture LAVC, pour la reconstruction

	d'événements d'ALEPH en parallèle
FASTBUS	Un standard international de système de contrôle et d'acquisition de données utilisé dans ALEPH
Générateur de stub	(Stub generator) L'un des utilitaires de RPC qui transforme le fichier de définition d'interface en "stubs"
GKS	(Graphic Kernel System) Un standard graphique international
HBOOK	Logiciel standard de manipulation d'histogramme au CERN
HCAL	(Hadron CALorimeter) Le Calorimètre Hadronique, un type de détecteur de particules
IDA	(Interactive environnement for Data Analysis) Un Logiciel interactif développé à SLAC
ISO	(International Standards Organisation) Une organisation internationale de normalisation
ITC	(Internal Tracking Chamber) La Chambre à Traces Interne, une partie du détecteur ALEPH
JULIA	Logiciel de reconstruction d'ALEPH
KAL	(Kinematic Analysis Language) Un système interactif utilisé dans ARGUS, une expérience de physique de particules à DESY, Hamburg, RFA
LAVC	(DEC Local Area VAX Cluster) Ensemble de stations de travail type VAX, liées par le réseau local Ethernet avec disques communs
LEP	(Large Electron Positron collider) Un collisionneur de particules au CERN

Macroprimitive	Primitives graphiques utilisées dans PIGAL
OSI	(Open System Interconnection) Un modèle en 7 couches recommandé par ISO pour que différents systèmes (matériels, logiciels) puissent s'interconnecter
Partage de données	Un modèle de communication entre plusieur processus
Passage de message	Un modèle pour une communication point-à-point
PAW	(Physic Analysis Workstation) Un outil d'analyse statistique et de présentation de données, développé au CERN
PCA	(Performance and Coverage Analyzer) Un logiciel de DEC pour mesurer la performance des logiciels
PIGAL	(Prolog Interactive and Graphic for Aleph analysis) Un logiciel d'analyse interactive de données pour ALEPH
POT	(Production Output Tape) Fichier sortie de JULIA, contient les événements reconstruits
Primitive	Sorties graphiques : ligne, cercles, cubes, etc.
Producteur	Voir Consommateur
Programmation distribuée	Une méthode de programmation pour réaliser les applications réparties
PROLOG	(PROgrammtion en LOGique) Un langage de programmation développé pour des applications d'intelligence artificielle, utilisé dans PIGAL
Protocole de communication	Ensemble de règles respectées par tous les composants de réseau pendant la communication
REASON	(Realtime Event Analysis Workstation Project) Un système développé à SLAC (Stanford Linear Accelerator Center), stanford, California, USA

Rendez-vous	Un mécanisme de communication bidirectionnelle
Répertoire d'événements	(Event Directory) Un fichier contenant les informations (numéro d'événement, run...) des événements sélectionnnés
RPC	(Remote Procedure Call) Un mécanisme de communication bidirectionnelle, utilisé dans PIGAL
RPCL	(RPC Language) Langage de description pour l'interface Client/Serveur de RPC
SASD	(Structured Analysis and Structured Design) Méthodes d'analyse et de conception de logiciel utilisées dans ALEPH
Sélecteur	Une entité spéciale créée par le module de sélection de PIGAL
SGR	(Structured Graphics Routines) Un logiciel graphique utilisé par les stations VAX
Stub	Elément de remplacement utilisé dans un programme, ensemble de subroutines d'interface produit par le compilateur de RPC
TIP	(Table Interaction and Plotting) Un logiciel interactif interfaçant à ADAMO, basé sur PAW
Tolérance aux erreurs	Un mécanisme pour récupérer le système en cas de défaillance partielle
TPC	(Time Projection Chamber) Chambre à Projection Temporelle, un type de détecteur de particules
Transaction atomique	Un mécanisme pour garantier la cohérence de base de données ou de fichier partagé dans une application répartie

UIS	(User Interface Software) Un logiciel de DEC pour piloter les graphiques et les opérations de fenêtres
X-WINDOW	Un logiciel standard portable pour contrôler la visualisation du graphique
ZEBRA	(Data Structure Management System) Un gestionnaire de mémoire développé au CERN

Cette thèse a été composée avec le logiciel MacWrite II (Version 1.1) pour la texte et le logiciel MacDraft (Version 1.2b) pour les figures, sur l'ordinateur Macintosh SE de Apple Computers.

Les polices de caractères choisies sont les suivantes :

Texte	:	Palatino 10,5 point
Commandes, Programmes	:	Courier 10 point
Têtes de chapitres	:	Helvetica gras 21 point
Têtes de sections	:	Helvetica gras 18 point
Têtes de sous-sections	:	Helvetica gras 15,5 point
Explications des figures	:	Geneva 8 point

RESUME

L'analyse de données des expériences de physique des particules dépend largement de l'informatique. Le travail décrit dans cette thèse vise à introduire de nouveaux outils informatiques dans les domaines de : logiciel graphique, langage de programmation, méthodologie de conception, technique de réseau local pour l'analyse de données.

Un environnement d'analyse basé sur une architecture distribuée est mis en oeuvre. Les composants du système sont répartis sur un réseau local et se synchronisent sous le contrôle d'un programme principal réalisé en langage PROLOG, ceci afin d'adapter l'analyse de données aux besoins de l'expérience ALEPH au LEP.

MOTS CLES

Analyse de données, Analyse interactive, Prolog, Graphique, Architecture du logiciel, Système réparti, Communication entre processus

ABSTRACT

Data analysis for high energy physics experiments depends widely upon computer science and technology. The work described in this thesis intends to introduce some new developments : graphic software, programming language, system design methodology and local network into the data analysis domain.

An analysis environment based on a distributed architecture is realized. The system components are connected together by a local network and are synchronized with each other under the control of a monitor written in PROLOG in order to satisfy the requirements of data analysis for (LEP) experiments.

KEYWORDS

Data analysis, Interactive analysis, Prolog, Graphic, Software architecture, Distributed system, Remote procedure call, Interprocess communication