Introduction au Framework

Laurent Duflot - LAL

I) introduction

II) survol

III) ProcessGroup

IV) exécution

VI) création d'un package

IV) autres aspects

Introduction

- Framework: cadre de l'analyse ou de la recontruction
  → ouverture des fichiers d'input/output, des fichiers RCP
  → configuration de l'environnement
  → "appel des routines utilisateur" *(pour parler FORTRAN -)*
- Package: code utilisateur ou code standard D0 pour créer/analyser/reconstruire des événements.
- RCP: Run Control Parameters
  pour contrôler l'exécution
  - du Framework, i.e. quels Packages appeler
  - des Packages standards du Framework (fichiers d'entrée/sortie, impressions...)
  - des Packages utilisateur (par exemple, jets de cône, rayon des cônes, seuils en énergie...)
- Hooks: (ou Interfaces) points d'accrochages du Framework. Il donne la main à l'utilisateur à des moments précis au cours du processing d'un événement.

ATTENTION:

- ceci est une présentation simplifiée du Framework.
- certaines choses peuvent changer car nous en somme à la toute première version

Exemples en fortran

- ALEPH:

  Dans ALEPH, le programme d'analyse s'appelle ALPHA et définit des routines utilisateur:

  - QUIBOS: changer dimension common BOS
  - QUIHIS: changer dimension common HBOOK
  - QUINIT: initialisation exécution
  - QUNEWR: pour chaque nouveau run rencontré
  - QUEVNT: pour chaque événement
  - QUTERM: à la fin de l'exécution

  Chaque utilisateur doit explicitement écrire la routine QUEVNT et faire appel aux packages standard et à ses propres routines. Le programme doit être modifié chaque fois que l'on veut ajouter/retirer l'appel à un package.

- D0 RUNI:

  - On notifie les packages que l'on veut appeler (standards ou propre à l'utilisateur) au Program Builder (pbd) qui construit le fortran correspondant.
  - L'exécution des packages standards est contrôlée par RCP
  - On peut définir ses propres packages

→ on ne touche pas directement au code mais on écrit des packages, ce qui facilite l'échange de code et permet d'enlever/ajouter facilement des appels à des packages en relançant pbd après avoir changé ses RCP.

Survol

- Exécution avec des packages utilisateur et standards

  Les packages à utiliser et les conditions d'utilisation des packages sont définis par RCP.

  → fig 2

  → fig 3

- Création de package

  Ce package peut être un générateur d'événements, un algorithme de jet ou d'identification d'électron, une sélection d'événement…

  – Ecrire des classes C++ selon les spécification du Framework.

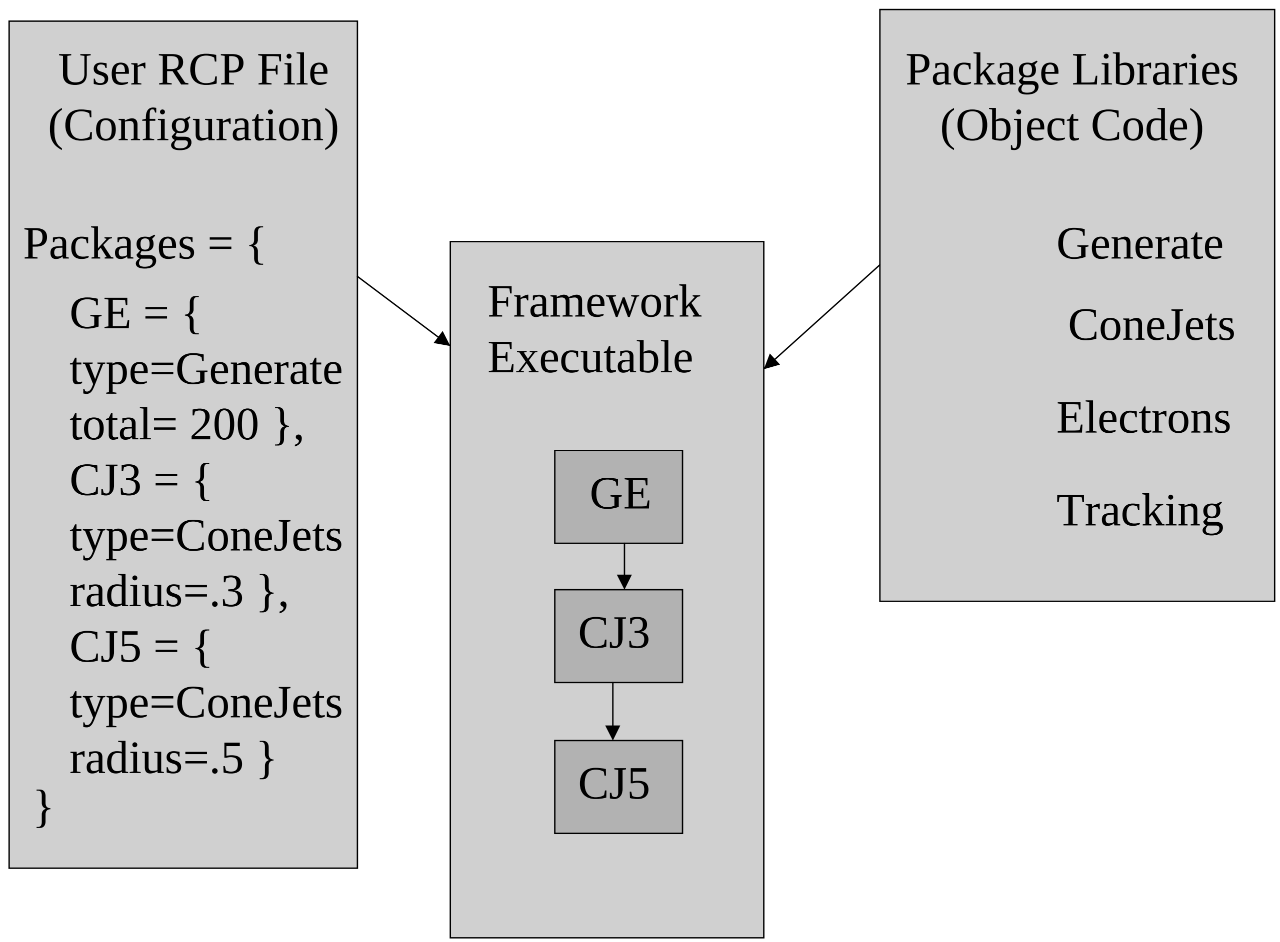
  – Le package doit avoir un (des) nom(s) et un numéro de version.

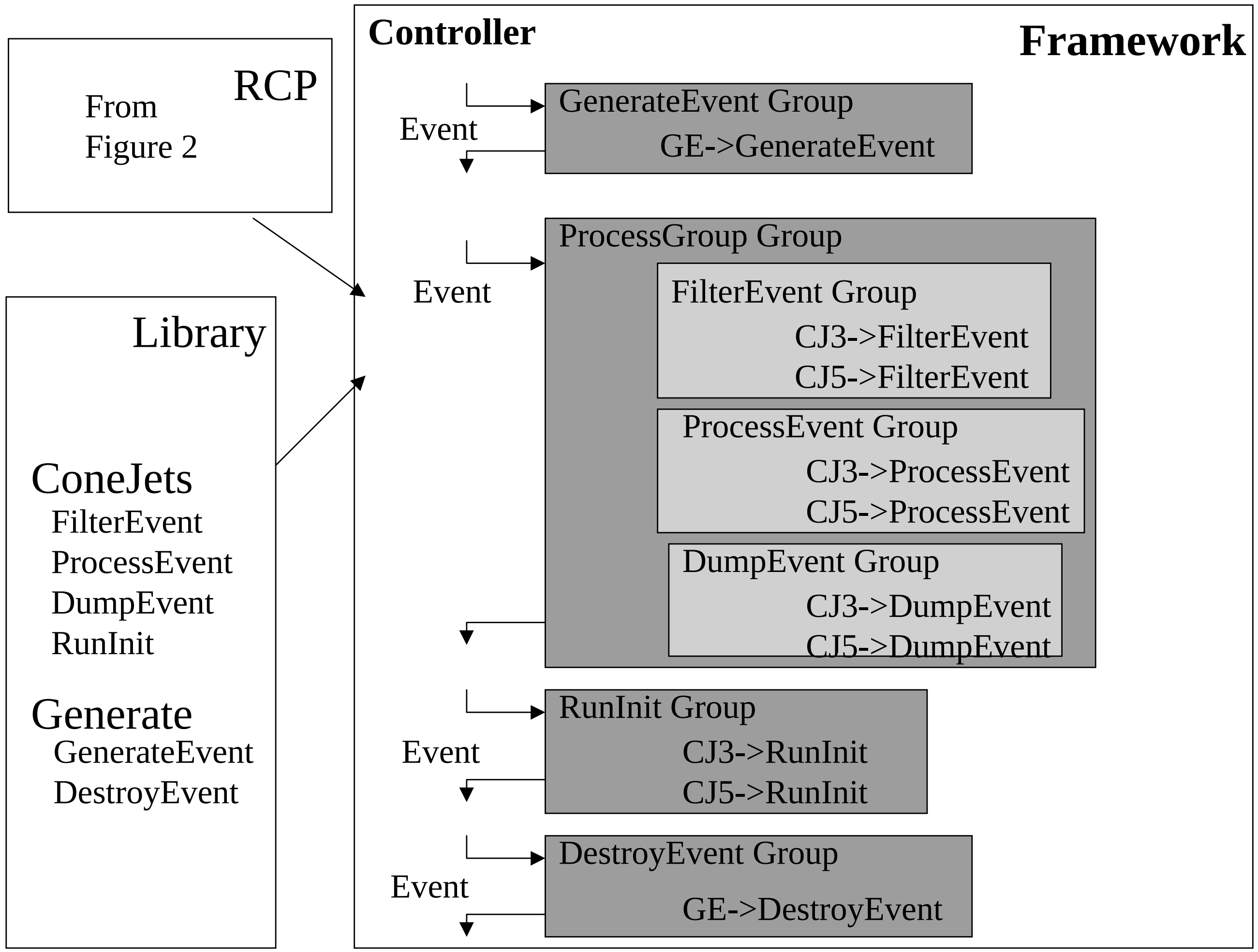
  – Les classes doivent s'enregistrer auprès du Framework pour être appelées par lui au bon moment. C'est un peu lourd mais en fait indispensable:

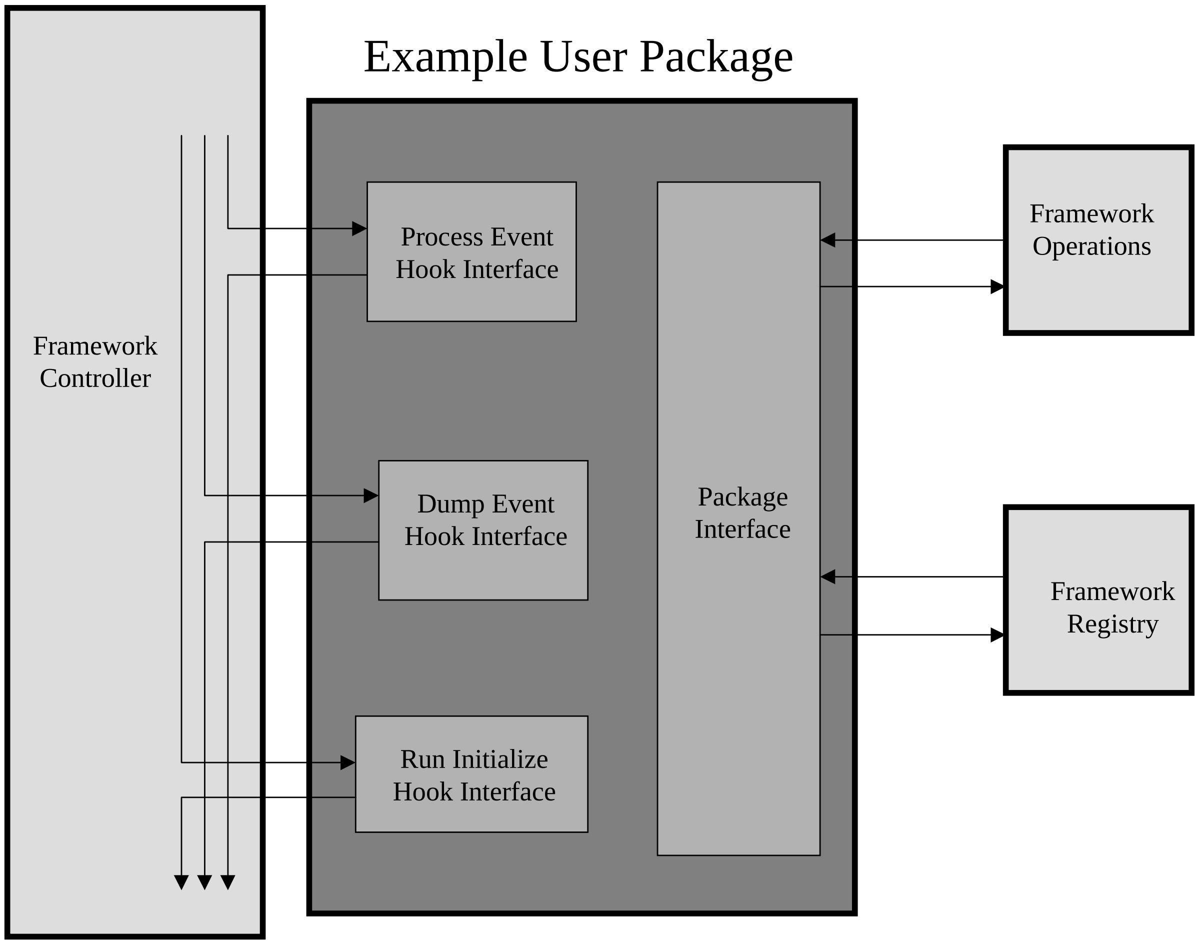
    → pour faire tourner plusieurs packages simultanément ou plusieurs version du même package

    → éventuellement faire tourner le code de plusieurs utilisateurs à partir d'un seul main (freight-train ?).

  → fig 1


User RCP File
(Configuration)
Packages = {
GE = {
type=Generate
total= 200 },
CJ3 = {
type=ConeJets
radius=.3 },
CJ5 = {
type=ConeJets
radius=.5 }
}
Framework
Executable
GE
CJ3
CJ5
Package Libraries
(Object Code)
Generate
ConeJets
Electrons
Tracking


RCP
From
Figure 2
Library
ConeJets
FilterEvent
ProcessEvent
DumpEvent
RunInit
Generate
GenerateEvent
DestroyEvent
Controller
Framework
Event
GenerateEvent Group
GE->GenerateEvent
Event
ProcessGroup Group
FilterEvent Group
CJ3->FilterEvent
CJ5->FilterEvent
ProcessEvent Group
CJ3->ProcessEvent
CJ5->ProcessEvent
DumpEvent Group
CJ3->DumpEvent
CJ5->DumpEvent
RunInit Group
Event
CJ3->RunInit
CJ5->RunInit
DestroyEvent Group
Event
GE->DestroyEvent


Example User Package
Framework Controller
Process Event Hook Interface
Dump Event Hook Interface
Run Initialize Hook Interface
Package Interface
Framework Operations
Framework Registry

ProcessGroup

- C'est un package spécial du Framework qui définit des Hooks où accrocher le code utilisateur.
- Les Hooks (Interface) sont codés de la façon suivante:

```
class INTERFACE: public Interface {
    INTERFACE(Context * hidden_info);
    virtual Results FONCTION(Event &)=0;
}
```

  La classe hérite de l'interface à implémenter et doit donc coder la FONCTION définie par l'interface.

- Les Hooks standards sont:

Interface	FONCTION	quoi/quand
Processor	processEvent(Event &)	analyse/reconstruction de l'evt
Filter	filterEvent(const Event &)	veut-on de cet evt ?
Dump	dumpEvent(const Event &)	impressions
RunInit	runInit(Event *)	nouveau run
RunEnd	runEnd(Event *)	fin de run
JobSummary	jobSummary()	fin d'exécution
Builder	buildEvent(Event &)	ajouter des chunks à l'evt
Generator	generateEvent(State &)	création d'un evt
	addDataToEvent(Event &)	
Destroyer	destroyEvent(Event &)	faire place nette

ProcessGroup

Nota Bene:

- Dès qu'un filtre rejette un événement, le processing de cet événement est interrompu.
- Créer un événement: aussi bien lire un événement à partir d'un fichier que créer un événement par un générateur.

Les Packages sont :

- ReadEvent : Lit un événement.
- NewEvent : Engendre un événement.
- WriteEvent : Ecrit un événement.
- DumpEvent : Impressions sur un événement.

Exécution d'un programme

- Tout est contrôlé par RCP: la liste et l'ordre des packages à appeler, les fichiers d'entrée/sortie, les paramètres pour les packages.
- RCP pour le framework

```
runframe.rcp :

string ProcessorPackage = "ProcessGroup"
string ProcessorHooks = "builder filter process dump"

// List of packages
string Packages = "p1 p2"

// RCP files for the packages.
RCP p1 = <ReadEvent.rcp>
RCP p2 = <mypackage.rcp>

// only call dump event hooks every this many events
int DumpPeriod = 1
```

- Il faut créer un fichier RCP par package appelé, par exemple

```
ReadEvent.rcp :

string PackageName = "ReadEvent"
string InputFile = "test.ds"
string InputFormat = "DSPACK"
int NumEvents = 0              // Maximum number of events to read (0 = all).
string ApplicationName = "ReadEvent"  // String passed to d0om_init.
```

ATTENTION: La syntaxe des RCP peut encore évoluer. . .

Création d'un package

Une classe doit être créée pour chaque interface à implémenter, elle doit hériter de Package et de la classe pour l'interface:

- header

```
#ifndef INC_CALCLUSTERRECO
#define INC_CALCLUSTERRECO
////////////////////////////////////////////////////////////////////////////////
//
// $Id: CalClusterReco.hpp,v 1.1.1.1 1998/05/11 22:15:14 serban Exp $
//
// File: CalClusterReco.hpp
//
// Purpose:  make clusters from calorimeter data
//
// Created:    20-APR-1998  Serban Protopopescu
// History:
////////////////////////////////////////////////////////////////////////////////

// Dependencies (#includes)

#include "edm/Event.hpp"
#include "caldata/CalTClusterChunk.hpp"
#include "caldata/CalZebraChunk.hpp"
#include "energycluster/EnergyClusterReco.hpp"
#include "energycluster/SconeClusAlgo.hpp"
#include "framework/Package.hpp"
#include "framework/hooks/Process.hpp"
#include "framework/Result.hpp"
#include "framework/Registry.hpp"

using namespace edm;
using namespace std;
using namespace fwk;
////////////////////////////////////////////////////////////////////////////////

class CalClusterReco: public Package, public Process
{

public:
```

```
  //  constructors/destructor
  CalClusterReco(Context* context);
  ~CalClusterReco(){;}

  // Overridden hook methods
  Result processEvent(Event &event);

  // Overridden package methods

  string packageName() const {return package_name();}

  static const string package_name(){return "CalClusterReco";}
  static const string version(){
    return "$Id: CalClusterReco.hpp,v 1.1.1.1 1998/05/11 22:15:14 serban Exp $";
  }

  //  make CalTClusterChunk objects
  void makeTClus(Event &event)const;
  //  get required input data
  CalDataChunk* getInputData(Event &event, float xyz[3],
     list<ChunkID> &parents)const;

private:
  EnergyClusterReco _eclusreco;
  SconeClusAlgo<CalTower, TowerAddress, CalTClusterChunk> _scone;
  string _algo;
  string _data;
  RCPID _rcpID;
};

#endif // INC_CALCLUSTERRECO
```

- code

```
////////////////////////////////////////////////////////////////////////////////
// File: CalClusterReco.cpp
//
// Purpose:  make clusters from calorimeter data
//
// Created:    20-APR-1998  Serban Protopopescu
// History:
////////////////////////////////////////////////////////////////////////////////

// Dependencies (#includes)
#include "calreco/CalClusterReco.hpp"
#include "edm/TKey.hpp"
#include "edm/THandle.hpp"
#include "framework/Registry.hpp"
```

```
/////////////////////////////////////////////////////////////////////////////////

static Registry<CalClusterReco> iamhere;

CalClusterReco::CalClusterReco(Context* context):
  Package(context), Process(context), _scone(), _eclusreco()
{
  out() <<" Initializing CalClusterReco package"<<endl;
  RCP rcp=packageRCP();
  _algo=rcp.getString("algo_type");
  _data=rcp.getString("data_type");
  _rcpID=rcp.getRCPID();

  if(_algo == "scone") {
    _scone=SconeClusAlgo<CalTower, TowerAddress, CalTClusterChunk>
      (rcp.getFloat("coneSize"),
       rcp.getFloat("seedET"));
  }
}

//  process event
Result CalClusterReco::processEvent(Event &event){
  makeTClus(event);
  return Result::success;
}
//  make CalTClusterChunk
void CalClusterReco::makeTClus(Event &event)const{

    //   get calorimeter data
    list<ChunkID> parents;
    float xyz[3];
    CalDataChunk* dataptr=getInputData(event, xyz, parents);
    CalData* calptr=dataptr->getCalData(xyz);

    if(_algo == "scone") {
      //   create a tower cluster chunk
      EnvID envid;
      CalTClusterChunk* tchunkptr=
      new CalTClusterChunk(_rcpID, envid, parents);
      auto_ptr<CalTClusterChunk> autochunk(tchunkptr);
      // make cluster collection for towers
      list<const CalTower*> tlist;
      calptr->getTowers(tlist);
      _scone.makeClusters(&_eclusreco, tlist, tchunkptr);
      insertChunk(event, autochunk);
    }
}

CalDataChunk* CalClusterReco::getInputData(Event &event, float xyz[3],
   list<ChunkID> &parents)const{
```

```
  if(_data == "zebra"){
    const TKey<CalZebraChunk> calKey;
    THandle<CalZebraChunk> calchunk=calKey.find(event);
    CalZebraChunk* ptr=calchunk.ptr();
    SimpleVertex v=ptr->getVertex();
    v.position(xyz[0], xyz[1], xyz[2]);
    ChunkID parentid=ptr->chunkID();
    parents.push_back(parentid);
    return ptr;
  }

  if(_data == "MC"){
    const TKey<CalDataChunk> calKey;
    THandle<CalDataChunk> calchunk=calKey.find(event);
    CalDataChunk* ptr=calchunk.ptr();
    ChunkID parentid=ptr->chunkID();
    parents.push_back(parentid);
    // temporary
    xyz[0]=0;
    xyz[1]=0;
    xyz[2]=0;
    //
    return ptr;
  }
  return 0;
}
```

- RCP

```
// Author: Serban Protopopescu
// Date:   5/8/98
//
// $Id: CalScone0710.rcp,v 1.1.1.1 1998/05/11 22:15:14 serban Exp $
//
//  read file generated from run 1 Zebra data
//  make clusters of cone size 0.7
//  use simple cone algorithm (SconeClusAlgo) with seed 1.0 GeV

string PackageName = "CalClusterReco"
string data_type = "zebra"
string algo_type = "scone"
float coneSize   = 0.7
float seedET     = 1.0
```

autres aspects

Il y a d'autres aspects du Framework dont je n'ai pas parlé:

- Status et Generator
- Context
- groupes de packages

  Par exemple, je veux pouvoir sélectionner simultanément les candidats trileptons et squarks/gluinos en faisant appel à ces deux packages définis par le groupe NP. Le package trilepton défini des filtres sélectionnant le trigger $e + \not{E}_T$ et les squarks le trigger $jets + \not{E}_T$. Mais dès qu'un filtre répond faux, l'événement n'est plus analysé. La solution est de mettre les deux packages dans des groupes de package différent car un filtre invalide le processing d'un événement *pour le groupe de package dans lequel il est appelé*.

  Comment faire en pratique ?????