Introduction à l'environnement software de D0

Laurent Duflot

- La commande setup
- Introduction au framework: le programme d'analyze de D0.
- Les RCP
- Utilisation du code existant.
- Développement de code dans D0.

gestion des produits logiciels

- Les logiels sont regroupés en produits: emacs, python, perl,....
- Ces produits sont gérés par ups/upd:
  - ups: gère une base de donnée des produits
  - upd: permet la mise à jour de la base de donnée en récupérant des versions nouvelles à FNAL
  - setup produit [version] [–q qualifiers]
    - setup emacs
    - setup d0cvs accès au rep. cvs de D0
    - setup D0RunII t00.74.00   permet l'utilisation de la release 74 du soft D0
  - prise en compte des dépendances: un setup peut en cacher 10 autres...

Introduction au framework

- Le framework est le cadre logiciel de la génération, de la reconstruction et de l'analyze:
  - ouverture des fichiers d'input/output
  - gestion des fichiers RCP
  - configuration de l'environment
  - appel des 'routines' utilisateur
- RCP: run control parameter
  - un fichier RCP pour controler l'exécution du framework
  - des fichiers RCP pour définir les paramètres des algorithmes (e.g. quels jets, quelles coupures ?).

- Un package du framework est une classe qui met en oeuvre un ou plusieurs points d’accrochage ’hook’.
- Le framework définit des points d’accrochage sur lesquels on peut se brancher pour effectuer des actions spécifiques à un moment donné dans le processing d’un événement.
- Un package du framework met en oeuvre un hook par héritage de la classe hook correspondante. Cela force à écrire une ou des méthode(s) définie(s) par la classe hook.
  - par ex., le hook Process force à définir la méthode processEvent(edm::Event& event); qui donne accès au contenu de l’evt.
- Un Event contient des Chunks de donnée (~ bank).

les hooks

- Generator
  - Construit un evt.
- Merge
  - Merge des evts de divers generators.
- Builder
  - Modifie evt.
- Filter*
  - Décide si le processing de l'evt se poursuit.
- Process
  - Modifie l'evt.
- Analyze*
  - Analyse l'evt.
- Dump*
  - Produit un dump (ascii) de l'evt.
- RunInit, RunEnd, JobSummary

* Ne peut pas modifier l'evt.

Ecriture d’un package

- hériter de la classe Package.
- hériter du/des classe(s) hook.
- Ecrire les méthodes requises:
  - contructeur avec un arg Context*
  - static const string version();
  - static const string package_name();
  - string packageName()
  - méthodes requises par le(s) hook(s)
- Ecrire une classe d’enregistrement du package auprès du framework pour qu’il puisse appeler notre package.

MyPackage.hpp

```
#include "framework/Package.hpp"
#include "framework/hooks/Process.hpp"

class MyPackage : public fwk::Package,
                  public fwk::Process
{
public:

  // Constructors/Destructors

  MyPackage(fwk::Context* context);
  ~MyPackage();

  // Overridden hook methods.

  fwk::Result processEvent(edm::Event& event);

  // Overridden Package methods.

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

  // Static methods.

  static const std::string package_name() {return "MyPackage";}
  static const std::string version();

private:

};
```

classe Package

classe hook

constructeur

méthode de hook

méthode requise par Package

méthodes statiques requises

MyPackage.cpp

```
using namespace fwk;

// Constructor

MyPackage::MyPackage(fwk::Context* context)
{...}

// Destructor

MyPackage::~MyPackage()
{...}

// Hook override.

Result processEvent(Event& event)
{...}

// Version method.

const string version()
{
  return return "$Id:"
}
```

ajouter:

#include « framework/Registry.hpp »

FWK_REGISTRY_IMPL(MyPackage, « Name: $ »

RegMypackage.cpp

```
#include « framework/registry.hpp »
using namespace fwk;


FWK_REGISTRY_DECL(MyPackage)
```

Packages standards

- Il existe des packages déjà écrit:
  - ReadEvent
    - Lit des evts sur fichier.
  - NewEvent
    - Crée un evt vide.
  - MergeEvent
    - Merge evts.
  - DropChunks
    - Permet d'éliminer des chunks.
  - WriteEvent
    - Ecrit des evts sur fichier.
  - DumpEvent
    - Produit un listing avec le contenu des chunks sélectionnés.

Le fichier RCP du framework

liste des hooks

```
string InterfaceName = "process"
string Interfaces = "generator decide builder filter modify \
  analyze process dump jobSummary runEnd runInit"
string Flow = "generator decide builder filter modify process \
  analyze dump"
```

ordre des hooks

```
// -------------------------------------------------------------------
// Packages

string Packages = "p1 p2 p3 p4"
```

ordre des packages

```
RCP p1 = <ReadEvent.rcp>
RCP p2 = <MyPackage.rcp>
RCP p3 = <WriteEvent.rcp>
RCP p4 = <DumpEvent.rcp>
int DumpPeriod = 1
```

les fichiers rcp des packages

MyPackage.rcp

obligatoire pour tous les packages

```
// Package Name
string PackageName = "MyPackage"

// Package rcp parameters
```

liste dépendante du package:

float x = 0.1
int i = 2
bool die = true
RCP params = <MyPackage parameters>

Les RCPs

- Les fichiers RCP servent à définir des paramètres int, float, double, bool, RCP et des vector<> de paramètres.
- Un fois le fichier lu, il est représenté par un objet de type edm::RCP qui a des méthodes d'extraction des paramètres
  - par ex: float getFloat(«nom »)
- Les RCP sont définis par le couple <nom–package nom–fichier>, ce qui suppose une structure de directory .../package/rcp/...
    - ici package est un package cvs
- Les RCP sont entrés dans des BD
  - globale/readonly pour une release
  - personnelle/read–write

Une session de travail

- Pour pouvoir travailler, même s'il s'agit d'exécuter du code déjà compilé, il faut mettre en place une structure de directory et faire référence à une release du code D0.
  - cd zone–de–travail
  - setup D0RunII t00.74.00
    - on se base sur la release test 74
  - newrel –t t00.74.00 travail
    - crée un directory travail avec la bonne arborescence: en particulier les dir lib bin et tmp qui contiendront les librairies, binaires et fichiers temporaires dans des sous directory dépendant de l'architecture (systeme+compilateur). Un fichier GNUmakefile est aussi créé.
  - d0setwa
    - indique que ce dir est la base des rcp personnels.

Faire tourner un exécutable de la release

- cd /scratch/duflot
- tcshd0 –l

à faire, temporairement, au CC

- setup D0RunII t00.74.00
- newrel –t t00.74.00 dev74
- cd dev74/
- d0setwa
- mkdir local; mkdir local/rcp
- cp $D0RUNII_DIR/d0reco/rcp/D0recoReadEvent.rcp local/rcp/
- cp $D0RUNII_DIR/d0reco/rcp/runD0reco.rcp .
- ln –sf $D0RUNII_DIR/bin/Linux2–KCC_3_3/D0reco_x .

exemple: d0reco

- xtagein –V wv0011 –q 39 –s 100 sim.electron_50GeV
- Changer la ’string InputFile’ dans local/rcp/D0recoReadEvent.rcp pour mettre le nom du fichier xtagé. Mettre int NumEvents = 5
- Changer dans runD0reco.rcp
  - RCP read = <d0reco D0recoReadEvent>
    - en
  - RCP read = <local D0recoReadEvent>
  - retirer write de la ligne string Packages = » »
- $D0RUNII_DIR/mc_exam/bin/make_link.csh
- ln –sf $D0RUNII_DIR/tau_hmchisq/test/tau_hm_s.dat .
- ln –sf $D0RUNII_DIR/tau_hmchisq/test/tau_hm_b.dat .
- ./D0reco_x –rcp runD0reco.rcp

Organisation software

- CVS:
  - le soft est organisé en packages de petites tailles (~150 packages actuellement pour le L3 et le offline). La maintenant du code est assurée par CVS. Chaque release du code D0 fait référence à un tag de chaque package. Il y a environ une release test par semaine et quelques releases de production par an.
- SRT:
  - Software Release Tools contrôle la compilation et le link. C'est un ensemble de fragments de makefile donnant accès de façon standard aux librairies (fortran, C++, cern, CORBA, etc).

- CTBUILD:
  - L’écriture des makefiles avec SRT restant assez complexe, un niveau supplémentaire a été introduit, même si tous les packages n’utilisent pas (encore) CTBUILD. L’organisation des éléments du package est décrite dans une série de fichiers de configuration et CTBUILD se charge de la traduire en fragment de makefile dans le cadre de SRT.
  - CTBUILD est la façon recommandée de gérer la compilation des packages, je me concentrerai donc sur la description des fichiers de CTBUILD.

Organisation d'un package CTBUILD

- On crée le squelette d'un package basé sur ctbuild par la commande
  - ctnewpkg –blodi monpackage
- L'arborescence suivante est créée sous le dir monpackage:
  - GNUmakefile: contrôle la compil.
  - monpackage: pour les header
  - src: pour les sources
  - bin: pour les sources des exécutables
  - test: pour les sources des tests du package
  - doc: pour ... la doc :–)

Les components

- Un package est éclaté en composants (component)
- A chaque component correspond une classe.
- Un component est formé de 3 parties:
  - interface publique: myfile.hpp
    monpackage
  - implémentation: myfile.cpp src
  - test de component: myfile_t.cpp src
- Le test de component est sensé exercer toute l'interface publique de la classe. Permet:
  - de trouver les pb plus rapidement
  - un contrôle de qualité

src/COMPONENTS

- Les noms (sans les extensions !) des components sont listés par ordre de dépendence.
- Les fichiers *.cpp seront compilés et ajoutés à la librairie qui portera le nom du package.
- les fichiers *_t.cpp sont compilés, linkés puis executés pour tester le component. Le code de retour du test conditionne le succès.

src/OBJECT_COMPONENTS

- Contient la liste des components qui doivent être compilés comme des fichiers objet hors de la librarie.
- Essentiellement utilisé pour les fichiers d'enregistrement auprès du framework.
- Il peut exister un fichier similaire D0OM_COMPONENTS qui contient la liste des fichiers pour les classes persistentes.

LIBDEPS

- Doit lister les libraries dont le package dépend directement (i.e. pour lesquels on a fait des ’include’).
- Avec les fichiers LIBDEPS des libraries dont on dépend, cela permet de reconstruire toute la chaîne des dépendances et de déterminer l’ordre de compilation des packages.

LIBRARIES

- Un fichier qui contient la liste des librairies nécessaires pour construire les exécutables.
- Pour l'instant, la liste de toutes les librairies est nécessaire. A terme, les fichiers LIBDEPS seront utilisés pour construire cette liste.
- Il y a un fichier LIBRARIES dans le directory bin (construction de l'exécutable) et dans le directory test (construction du test intégré).

Tests intégrés

- Une fois testés individuellement, il reste à tester les components ensemble, c'est le test intégré.
- Le fichier ITEST contient la liste des tests intégrés.
- En général dans le directory test, acompagné de fichiers LIBRARIES et OBJECTS (liste des .o nécessaires).

bin/BINARIES

- Contient la liste des exécutables à créer. Il doit y avoir un fichier .cpp du même nom à compiler.
- Accompagné dans le directory bin d'un fichier LIBRARIES et d'un fichier OBJECTS.

Un exemple concret

- Retourner dans le directory de développement
- setup d0cvs
- addpkg –h kt_jets
  - ramène la version head du package kt_jets
- Examiner le contenu du package
- compiler par
  - gmake

Remarques

- Ceci n'est qu'un point de départ pour le développement de code dans D0...
- Le système est assez complexe à maîtriser.
- Il est indispensable de comprendre le framework et les RCP. Voir la doc du framework dans framework/doc.
- L'écriture de package est la base de l'analyse: on ne peut pas y échapper!
- Voir mon talk sur SAM pour l'accès aux données a FNAL.