ScriptRunner Implementation Notes

Moacyr H.G Souza
for the L3 Filtering Group

last revision : $april 10^{th}, 1999$

1 Introduction

The main components of the L3 Filtering, mostly derived from RunI naming convention, are (see [ref 1,2] for the complete description of *Filters* and *Tools* functionality)

1. *ScriptRunner* is the driver of the DØ Level III Trigger Filtering. It implements all the logics to honor the Trigger List on duty. ScriptRunner will run as a stand-alone executable, steered by the L3 On-line Framework. As such, it will interact with the L3 Framework through L3 hooks - some will be the same as for off-line, as Begin-of-Run, Event processing , etc. - and other specific ones will pass directives, parameters and/or ask for some services, such as monitoring data (see [ref 3]). Its main task is to parse the *Trigger List*, and, for each event, to execute the one or more L3 *Filter Scripts* associated with each *L1/L2* trigger bit that fired for this event, call *Filters* and *Tools* in the appropriate order, returning a trigger decision (*passed, failed*) to the L3 Framework .
2. *Tools* - reconstruct physics objects. They are algorithms, of which there can be multiple instances distinguished by different Reference Sets (*RefSet*) of parameters. Since most of the available processing time is spent in the tools, and tool instances can be called multiple times, one of their important features is that they will *cache* all their results. Tools can call others Tools to make their calculations and this must be possible without external intervention.
3. *Filters* - call Tools to obtain reconstructed physics objects and check these against a set of parameters (Parameter Set - *Parset*). They return a *Failed or Passed* status.
4. *Filter Scripts* - correspond to a L3 Filtering Trigger Bit and are basically logical *ANDs* of several *Filters*. They are grouped following each L1/L2 trigger bit: for each L1/L2 bit that passed, all associated L3 Filter Scripts will be tried. A given *Filter* can be a part of several *Filter Scripts*. An event is kept (passed) if at least one Filter Script is completed with success (all *Filters* associated with it passed). However, all Filter Scripts, for a given L1/L2 bit, will be tested, no matter if the event has already passed or not, both for monitoring and rates purposes.

The design of the ScriptRunner implementation was based on some tight constraints:

- Speed - L3 has make a decision in about 50 - 70 msecs per event. The majority of the time should be allocated to re-constructors (Tools), so that the filtering framework (ScriptRunner) should not take more than a tiny slice of this time. All (of the Tools') reusable intermediate results must be locally cached.
- Reliability - That means simplicity and straightforwardness - we can't afford to loose data because of some fancy/obscure faulty code! This was used largely even under expenses of some readability and "user-friendliness".
- In addition, we took into account some wanted features/constraints, which were already pointed out in several previous documents. They are the integration with the overall L3 environment - General L3 Framework, Microsoft Windows NT OS & Visual C++ compiler and the integration with the DZero General Framework. Tool developers should be capable of switching from Off-Line to On-Line environment with minimal harm, and compatible with the above constraints.

The main ingredients of the L3 Filtering design are:

- Tools (and Filters) are implemented as *Singletons,* allowing automatic local caching of both the return status as well as their results. They are keyed by their RefSets, not by their type: Ele_Loose and Ele_Tight are different Tools of the same type, L3TEle.
- A **Linked List** pattern was found to be a very convenient way to implement the Trigger Logic. Appropriate iterators have all the functionality of calling Filters (which call Tools) and deal with whether and what to write to the Event (using the EDM interface) and to keep statistics. Filters and Tools are left to do very precise things.

In the following sections we will discuss the implementation details of these components.

2 The Filtering Framework

The Level 3 logic is implemented *on the fly* for each Run (i.e. for each TrigList) by a *Linked List* class and one *Iterator* class : the so called *External Iterator*, that performs, at run time, the actual FILTER DISPATCHING. As usual, the Iterator class is made a *friend* of the Linked List class. In most senses, the External Iterator will be the *BOSS* of the overall process: both by having detailed knowledge of each step of the process and by being declared *a friend class* by the other classes. Its design is open and flexible enough to accommodate additional future functionality. The *ScripRunner* class acts as the interface of the L3 Framework to the External Iterator class.

2.1 The *Linked List* (*Execution Tree*)

The Linked List is built from parsing the TrigList each time a new TrigList is downloaded or upon request of the L3 Framework (see below). For each element (*request*) inside the TrigList , a *link node* is created. As usual, a node will know the pointer to the next node. At the *link node* instantiation, a new *Filter* object of a type keyed by its name is created (instantiated). So, a *link node* will have only the following two functionalities: to hold a pointer to the next element of the list and to hold a pointer to a "Filter" associated with itself (the node). "Filters" can be either a *Filter*, a *Filter Script* or a "fake" node - see below. Figure 1 shows how an *Execution Tree* is made. *PLEASE NOTE that this scheme DOES NOT represent any kind of execution path or class diagram. That is, each node DOES NOT CALL the next node.* The *Iterator* will traverse the List, at run time.

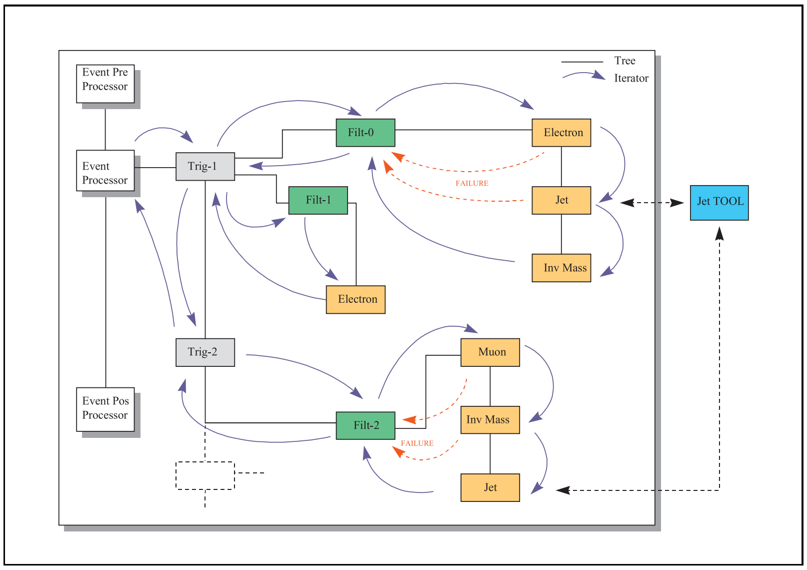

Figure 1: Execution Tree

In this figure we have the following components:

- *Trigs* - represent L1/2 Trigger bits. They are *link nodes* pointing to *Fake Filters*, that is, *nodes* that have their functionality in implementing the Trigger Logic, but point to *do-nothing Filters*, that is, they always return *true*. They will be traversed by the *External Iterator*. They are also the starting point of each branch. Provision was made for them to also have *Filter functionality*, that is, to point to actual Filter objects, as they can be useful to implement overall Filters, such as L2 simulators, etc...
- *Filt-n* - that is , Filt-1, Filt-2... Those are *nodes* pointing to the actual L3 Filter bits, the *Filter Scripts*. They can also, if wanted, point to actual Filter code. Note that the *Filter Scripts* will only have their status (passed, failed) by the end of the traversing of the branch. In the Linked List pattern, they have the *Mother node* functionality: *Child nodes* pointing to *Filters* will have them as their *return address point* in both cases of success or failure.
- *Filters* - have the *child* functionality. When traversed by the Iterator, their Filter code will be executed, returning a status. The Iterator will proceed , in case of success, until full completion or return to the *Mother*, in case of failure.
- *Tools* - as shown, Tools are not a part of the Execution Tree, they are called by Filters. Several Filters from several Filter Scripts can call the same Tool.

2.2 *External Iterator*

The External Iterator is instantiated by the ScriptRunner constructor. It parses the Trigger List to build an Execution Tree(Linked List) every time this is requested by ScriptRunner. At Event

Processing time, it is asked by ScriptRunner to initiate the *external traversing* of all L2 trigger bits. It has *loop-like* methods

```
void Begin();
bool IsFinish();
operator ++
```

that implement the *L3 Filtering Loop* over the L2 trigger bits:

```
for( It.Begin() ; It.IsFinish() ; ++It )
```

Only *both* active and fired L2 bits are considered. For each such L2 bit, it invokes the `Traverse`() method to traverse *this* particular branch (Filter Script).

So, the *External Iterator*'s charges are :

2.2.1 *Parsing*

The Iterator will parse a given Trigger List through the method

```
It.addL2Bits()
```

that is called by ScriptRunner each time the AddL2Bits hook is issued by the L3 Framework. This parser can recognize Tool directives:

```
ELE_LOOSE L3TEle(EMSEED=EM_SEED, TRACK=EM_TRACK, EMFR=0.90, likelihood=2.0)
```

In this example, a Tool of type *L3TEle* is declared with *RefSet* name ELE_LOOSE. It needs the results of other Tool instances, EM_SEED and EM_TRACK. The remaining are parameters specific to this instance of tooltype *L3TEle* .

Filter directives, in turn, will be parsed as

```
TRIG_79
filter test_79_0
L3FMarkAndPass key=Mark&Pass_10 pass_1_of_n=10. Stream=''monitor''
L3FEle key=ele1 refset=ELE_LOOSE number=1 MinEt=5.0 MaxEta=2.3
L3FEle key=ele2 refset=ELE_LOOSE number=1 MinEt=15.0 MaxEta=1.5

TRIG_79
filter test_79_1
L3FHt key=ht1 filter1=ele1 filter2=ele2 MinHt=30. MaxHt=1000.
```

In this example, the L1/L2 trigger bit "79" is declared with *two* branches (it is repeated), corresponding to one *Filter Script* each ("test_79_0" and "test_79_1", respectively). *Filters* are grouped under each *Filter Script*, they are keyed by "names" (ParSets) corresponding to the Tools (keyed by their RefSet) they need and some *cuts*.

Appropriate RCP files define the format for each Trigger List declaration, along with default values, if any, and upper and lower limits.

Both Tools and Filters have the pseudo-constructor method

```
MakeMe();
```

(see below).

When the Parser finds a *Tool* directive, it makes a instance of the Tool, keyed by the RefSet. It's flagged as a fatal error for two Tool directive to refer to the same RefSet (with exception of the "l3shop" test bank, see below). Tool pseudo-constructors are able to do two things:

1. Get their parameters through standard Parser methods and check them against RCP files for their integrity and meaningfulness.
2. Get pointer(s) to the Tool(s) they need, if any, with the appropriate RefSet(s). Of course, "lower level" tools (which might be called by other Tools) must be declared before the others, following the order of the calling. Otherwise, a fatal error is issued.

When the Parser finds a *Filter* directive, it creates a *link node*, as explained in the previous section, and makes a instance of this Filter, keyed by its name. Similarly, Filter pseudo-constructors are able to do two things:

1. Get their parameters through standard Parser methods and check them against RCP files for their integrity and meaningfulness.
2. Get pointer(s) to the Tool(s) they need with the appropriate RefSet(s). A fatal error is issued if this (these) Tool(s) was(were) not already instantiated.

All inconsistencies in the Trigger List format and Filter and Tool interdependency faults are detected at this Parsing level and the appropriate errors are issued. Fatal errors result in the ScriptRunner abort.

2.2.2 Branch Traversing

For each step of the *External Iterator* Loop the method `bool Traverse()` is called,which does the actual *Filter Dispatching* for each L2 trigger bit. It does so by knocking on each node's door. If this is a *Fake filter,* a *Unbiased Filter* (L2 unbiased, L3Mark&Pass 1_n, ...), the appropriate action is taken. When it finds a node that has a pointer to an actual Filter, it *EXECUTES THIS FILTER*. That is the way Filters are invoked in this scheme. This call is made by

```
_current->_GetPt()->DoThisFilter();
```

where *_current* is the address of the current *node*, *_GetPt()* returns the address of the *Filter* and *DoThisFilter()* is the bool method that returns the status of this *Filter's* execution , *success* or *failure*. Of course, DoThisFilter() will call the corresponding Tools (but let's stress this point: *ScriptRunner, through the External Iterator, does not see Tools at the execution pass.* Nevertheless, it has write access to the Tool area, by looping over the Tool register map [ref 2]). If the filter passed, the iterator gets the address of the next node and performs the very same procedure. Conversely, if the returned status is "failed", or the end of this branch is reached, it returns to the *Mother*. After the branch is traversed, the status is returned to the caller. In case of success, *ScripRunner* will invoke the *L3Chunk* generator. It will loop over all registered Tools and Filters and will write all L3 information in the Chunk (see [ref 2]).

2.2.3 Other Iterator methods

The main ExternIterator methods are:

```
 class ExternIterator {
public:
 ExternIterator ();
```

Note that the *Destructor*, when invoked, must deal with the deletion of all objects (*Tools & Filters*) created so far.

```
 ~ExternIterator() {}

 // this is the ''return'' status for one event
 bool Passed();

 // Method ''Traverse''
 // To traverse a given L2 trigger bit branch.
 void Traverse();

 // loop-like methods/operators
 void Begin() ;
 bool isFinish() const ;
 void operator ++() ;

 linkNode *_Current() const ;
```

An overall *Reset* method will steer cleanup before the next event. This is implemented as a loop over all registered Tools and over all registered Filters, with a call to each *Reset() method* . All Tools and Filters implement those methods [ref 2]

```
// Event by Event cleanup:
 void ToolsReset() const ;
 void FiltersReset() const ;
 void Reset();
 bool L3ChunkGen() ;
```

The following are methods called by ScriptRunner to honor L3 Framework hooks [ref3].

```
 //hook servicing methods
 bool BeginRun( bitset<MAXL2> & );
 bool EndRun( bitset<MAXL2> & );
 void Summary() const ;
//...

 // This method parses the Trigger List on duty. For each line, actions
 // are taken to instantiate Tools and Filters and methods are called
 // to build the execution Tree, on the fly. Called from the ScriptRunner
 //  takes a List name as argument.
 // Created: 05/05/98 Carmem Silva
```

```
 void AddL2bits( l3string );

 void DropL2Bits( bitset<MAXL2> &dropit );

 void Destroy();

private:
 bool passed;
 List triglist;
...
 // this is the event mask
 bitset<MAXL2> L2Eventbits;
 // this is the L2 currently active bits
 bitset<MAXL2> L2bits;
 // this is the array of l2bits
 int bitsort[MAXL2];

 // container classes
 // run statistics, by l2bit
 l3map<const char *, NodeStat > SumStat[MAXL2] ;
 // monitoring statistics
 l3map<const char *, FilterStat> Stat;

 // Those are traversing pointers
 linkNode *_excurrent;
 linkNode *_current;
...
}
```

3 Tools and Filters

For a detailed description of the Filter and Tool functionality, see [ref 1,2]

Tools are implemented as *Singleton* (Gamma et al book) objects : they are a sole Instance of each RefSet. That is, every instantiation of a given Tool keyed by a given RefSet results in the creation of a new object only if this object doesn't exist at this time. Otherwise, the address of the existing object will be returned. This enforce *caching* in a very straightforward way.

We defined a base class *bTool* that has as essential ingredients, two pure virtual methods

```
 virtual void MakeMe()      =0;          //     Pseudo constructor
 virtual bool DoThisTool() =0;          //     Actual invocation
```

which make mandatory that all sub-classes (the actual Tools) implement their own copy .

Any Tool instantiation registers its address in a *map* keyed by the REFSET *NAME* ! A "test shop" is also set up - *l3fshop*, allowing insertion, at the Trigger List level, of new test (not official) Tools, or overwriting of existing ones, without disturbing the main Tool implementation file.

The *Filter* objects are also implemented as *Singletons*. This is mainly for convenience since

Filters don't do calculations and Tools results are already cached. This also implies that we have a Filter registration for each instantiation, easing the implementation of the TrigList logic. Thus we have the following logic: during traversing, ScriptRunner *sees* only the External Iterator class, which *sees* only Filters in the same manner as Filters *see* Tools.

4 *ScripRunner* Skeleton

```
#include ''ScriptRunner/ScriptRunner.hpp''
#include ''ScriptRunner/options.hpp''
#include ''l3fparser/ParserParameters.hpp''
#include ''l3fresults/L3Chunk.hpp''
......

// need this to pass on the event location:
#include ''l3ftoolbase/unpackTool.hpp''

....

//----------

Framework registration ...

//----------
//ScriptRunner Constructor
ScriptRunner::ScriptRunner(fwk::Context* cc):
                          fwk::Package(cc),
                          fwk::Process(cc),
                          fwk::JobSummary(cc),
                          fwk::RunInit(cc),
                          fwk::RunEnd(cc) {

         cout << ''constructing package ScriptRunner: ''
                                    << endl;

// Instantiates ScriptRunner

   Clocking initialiation ...

  Set up ErrorLogging ...

        // instantiate the External Iterator:
        It = new ExternIterator();

     ......

 // the two directives below will correspond to addL2bits hooks (not yet implemented):
 l3string TriggerList0 = packageRCP().getString(''TriggerList0'');
 It->AddL2bits(TriggerList0 );

 //put more stuff on the execution tree (AddL2bits)
```

```
 l3string TriggerList1 = packageRCP().getString(''TriggerList1'');
 It->AddL2bits(TriggerList1 );

    .........
}

//----------
//Destructor
ScriptRunner::~ScriptRunner() {
      .....
       It->Destroy();
       delete It;
}

// ------------------------------------------------------------------------

//Begin Run hook:
fwk::Result ScriptRunner::runInit(const fwk::RunNumber &rno) {
     ....

   // ''thisl3bist is to be passed by the online hook
   bitset<MAXL2> thisl2bits(0xF);

   // this is the begin run handler
   if( ! It->BeginRun(thisl2bits) ){
     //  throw it!!

   }

          return fwk::Result::success;
}

// ------------------------------------------------------------------------
//Event processing hook:

fwk::Result ScriptRunner::processEvent(edm::Event& event) {

     //beginning l3filter timing for this event
     clock_t start= clock();

     // Tells the unpacker the event location
     unpackTool::evtLoc = &event;
     ...

     // External (Event) Traverse Loop
     for( It->Begin() ; It->isFinish() ; It->operator ++() ) {

           // Internal traversing
           It->Traverse() ;
            ...

     } // End External traversing
```

```
     if( It->Passed() ) {

         It->L3ChunkGen();
     }

        ....

     It->ToolsReset();
     It->FiltersReset();
     It->Reset();

       ....

     if( It->Passed() )
    return fwk::Result::success;

     else
    return fwk::Result::failure;
}

//-----------
//End Run hook:
fwk::Result ScriptRunner::runEnd(const fwk::RunNumber &rno) {

  bitset<MAXL2> thisl2bits;
     // this is the end run handler
   if( ! It->EndRun(thisl2bits) ){
     //  throw it!!
         ....
   }

          return fwk::Result::success;
}


// ------------------------------------------------------------------------
//  Summary : It->Summary() can be called by others hooks at any time,
//            actually means '' summary so far''

fwk::Result ScriptRunner::jobSummary()
{        ...
       It->Summary();
       ...
       return fwk::Result::success;
}
```

5 References

[1] D0Note 3266 ‘‘Description of D0 L3 Trigger Software’’, A.Boehnlein, D.Claes, M.Souza and G.Watts.

[2] D0Note 3630 ‘‘Description of D0 L3 Trigger Software Components’’, A.Boehnlein, G.Brooijmans, D.Claes and M.Souza

[3] ‘‘Level 3 interactions between L3 framework and ScriptRunner’’, A.Boehnlein, D.Claes, M.Souza and G.Watts.