ADVANCED FRAMEWORK MODULE STRUCTURES FOR THE IRIS DEVELOPMENT PLATFORM

Dima Mihai-Octavian, *Doctor of Physical and Mathematical Sciences, Leading Researcher*
V.V. Korenkov, *Doctor of Engineering sciences, Chief Researcher*
Joint Institute for Nuclear Research
(Russia, Dubna)

DOI:10.24412/2500-1000-2022-6-1-136-140

The project was realised in the 05-6-1119-2014/2023 "New Investigation Tools in Computational Physics" research theme, of the Meshcheryakov Laboratory of Information Technologies of the JINR.

Abstract. *The IRIS Development Platform delivers compilation, cross-project linkage and data fetch services, while allowing hot-swaps for performance comparison of the modules under development. While previous work was dedicated to the framework itself [1], the current paper details the structure of the AFM modules assembled under IRIS. The development arised from the need for advanced performance modules, that are standardised and independent of frameworks conceptually C++98 tributary. It provides an example based guide to scientists towards direct implementation of code in separate compilation model of templated C++11 technology (with a vision for conforming to Concepts^{TS}, allowing relatively facile upgrade to C++20).*
Keywords: *software development platform, C++11, shared object technology.*

Introduction. The IRIS platform originated from the software design under the ATHENA Framework [2] of the ATLAS experiment at CERN. In 2004 this platform was in its development stage and incurred considerable overhead when accessing data and running over untuned sections of code of the various sub-systems. The principal problem encountered was the very slow code compilation and linkage, a much faster framework being needed. Along this, the flexibility of allowing hot-swaps of software libraries among developers was also desired.

IRIS [1] answered positively both these demands, giving flexibility to developers in comparing their work, by routine hot-swapping and shared-object library creation – the shared object libraries being "un/mounted" at any point on the “beanstalk", presented in figure 1. This approach enhances code reliability and versatility. The “mounting” of routines on the framework is in a way similar to i-node attachment in populating a file-system.

Computing task

Unlike similar frameworks of extensive scope (GAUDI - [3]), the purpose of AFM is closer to the gcc compiler and programmer alike. Most developers in physics (and sciences in general) have a limited Computer Science background, and are conceptually F77 programmers even if they code in C++. This is the profile to whom AFM (and IRIS) reach out to.

A scientist typically develops a set of codes, that we will assume to be C++ classes. The AFM prototype needs to be one of these classes: encapsulated in its own namespace (requiring down-visibility onto other namespaces is uses) and its own library.

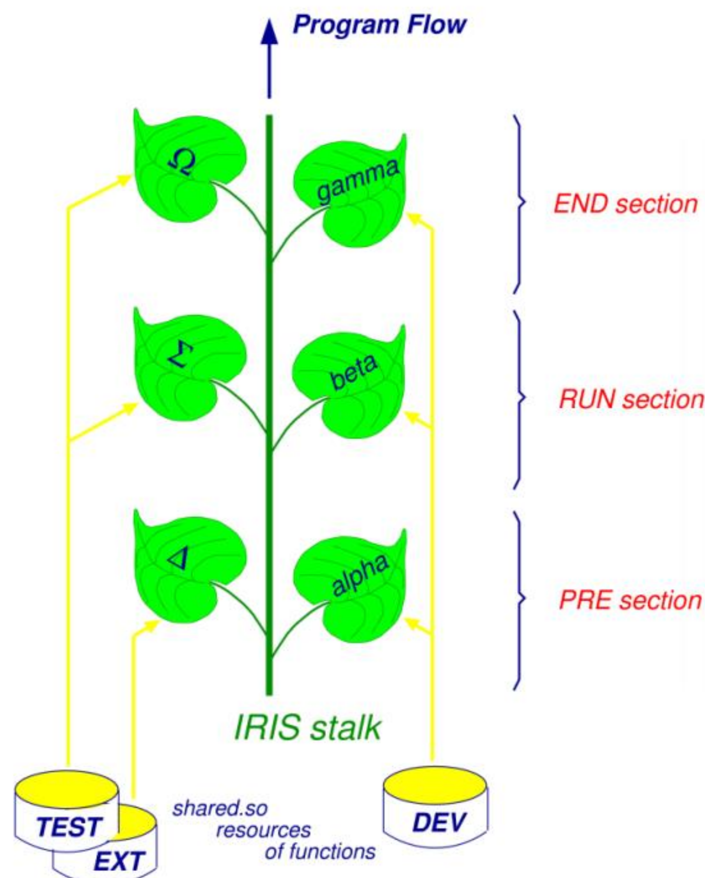

Figure 1. With IRIS work can be flexibly hot-swapped in/out among developers using shared object libraries – by "un/mount" at any point on a chain resembling a bean-stalk. The iris code compiles dev.so and test.so libraries and at runtime can open those for desired functions, or override them with functions from other ext.so libraries

The assembly of AFM's is to be performed by IRIS, the governing framework.

The software system we are addressing is not dedicated (on/off-line, trigger, reconstruction, etc). These are domain specific tasks that require user-dedicated attention, not general framework focus.

The AFM's will provide a skeleton for building software in a unitary way that is project compliant and advanced in performance.

The simplest unitary way to achieve this is with shared object technology. The operating system provides facilities for the use of dynamically linked shared libraries – therewith the external symbols referenced by the user defined in the shared library are resolved by the loader at load time. This significantly reduces the size of the executable, fitting it into cache, which is the important point. The downfall of the approach is that shared code is loaded into memory once in the shared library segment and shared by all processes referencing it. But the code scientists write is not system wide, hence the shared library segment will be in 1:1 with the user. Runtime performance is enhanced because the operating system is unlikely to page out shared library code even if used by just one user, on a multi-processor platform. The risk of page faults we estimate as low compared to constant page flipping of large static bundles.

From a performance viewpoint the "glue code" required to access the shared segment (ca. 8 machine cycles per reference) is outweighed by the slim line of the shared objects (vs. huge static bundles faulting cache pages by their sheer size). Our experience running in this mode is anecdotically very positive, however in the absence of a rigorous study in this respect we can only extrapolate our experience, along the guide lines mentioned. We cannot comment on the reduction in reference-locality. It may happen that for the full code the routines to touch are all over the vir-

tual address space in one library, hence the total number of pages to be accessed is significantly higher than if neatly ordered in one static bundle. Yet again, the (constant) high rate of page flipping when using static bundles kills the highly repetitive sections of code, which would otherwise be confined to the cache. Further more – modern machines and compilers highly optimize cache fluxes – advantaging work with dynamic objects.

Description of the *AFM's*

An AFM is a an example project. A shell exec addx asks for the name of the new project and its intended programming level. It subsequently creates the directory structure, the makefile and fills the include and src directories with example files that compile and run (of the respective programming level). The directory structure is shown in figure 2.

The programming levels are: *simple* (namespace + function), *medium* (namespace + non-templated complex number class), *m2* (namespace + templated SU(2) spinor class) and *advanced* (namespace + templated vec with resource).

```
lxplus719.cern.ch> ll
total 18
drwxr-xr-x. 2 modima it 2048 Mar 16 17:59 examples
drwxr-xr-x. 2 modima it 2048 Mar 16 17:59 include
drwxr-xr-x. 2 modima it 2048 Mar 16 17:59 lib
-rw-r--r--. 1 modima it 5190 Mar 16 17:59 makefile
drwxr-xr-x. 2 modima it 2048 Mar 16 17:59 obj
-rw-r--r--. 1 modima it  277 Mar 16 17:59 README
drwxr-xr-x. 2 modima it 2048 Mar 16 17:59 src
-rw-r--r--. 1 modima it  987 Mar 16 17:59 test.cc
lxplus719.cern.ch> █
```

Figure 2. The directory structure of an Advanced Framework Module follows a traditional layout, with the classes separated in the include and src directories (instantiation *.ie files also in src), for separate compilation model. The makefile takes the clean, libs, test and run commands

The examples are all set up and running, at their respective levels. The organizational effort beyond *simple* is considerable (CPX complex numbers have 2500+ instantiations, SU2 spinor scalars have 5700+, vec entails numerous re/cast details). Indeed, giving only a few simple operators and functions in the examples, they would have looked infinitely simpler, however this is intended as a full-project guide – meaning respectively the complete picture, which can be painfully extensive, the examples serving as roster for the full list of functionality that the programmer needs to implement.

```
libs: objects

objects: abx.hh abx.cc
        cd obj ; rm *.o ; g++ -std=c++14        \
                           -O3                   \
                           -z muldefs            \
                           -fopenmp              \
                           -fopenmp-simd         \
                           -floop-block          \
                           -ftree-loop-distribution   \
                           -floop-parallelize-all     \
                           -ftree-parallelize-loops=4 \
                           -ftree-vectorize      \
                           -fconcepts            \
                           -mavx2                \
                           -I ../include         \
                           -I ../../CPX/include  \
                           -c                    \
                           ../src/*.cc
```

Figure 3. Sniplet from an AFM makefile. Note the OMP-5.1 and (Intel AVX2) SIMD directives for parallelization and vectorization, as well as the allotment for Concepts^{TS}.

The makefile is relatively plain, figure 3 showing a sniplet thereof with OMP-5.1 parallelisation and (Intel AVX2) SIMD vectorization, as well as the allotment for Concepts[TS]. The compilation options can further be grouped into a symbol and used as $(SYMBOL), however this fancy scripting is besides the point of the makefile's functionality. The most of such fancy scripting is depicted in figure 4 - the separation and renaming of static vs. dynamic libs.

```
cd obj ; ar rs libabx4.a *.o
cd obj ; mv libabx4* ../lib
cd obj ; for filename in *.o;                   \
         do echo mv \"$$filename\" \"$${filename//.o/\-static.u}\"; \
         done|/bin/bash
 ... compile dynamic
cd obj ; g++ -std=c++14                         \
             -O3                                \.
              libabx4.so *.o
cd obj ; mv libabx4.so ../lib
cd obj ; for filename in *.o;                   \
         do echo mv \"$$filename\" \"$${filename//.o/\-dynamic.o}\";\
         done|/bin/bash
cd obj ; for filename in *.u;                   \
         do echo mv \"$$filename\" \"$${filename//.u/\.o}\";  \
         done|/bin/bash
```

Figure 4. Sniplet with some of the (minimal) scripting used in the AFM makefile (separation and renaming of static vs. dynamic libs). The important aspects are the inclusion of the relevant set of compilation options and forthcoming the ability of the user to adapt the makefile to include the packages that it depends upon (here as example CPX, complex numbers)

Integration with IRIS

The old iris executable expected the contributing "leaves" to be present in the running directory. This had to be modified, as the compilation function was delegated from IRIS to the AFM's themselves.

The new IRIS runs the "leaves" mounted on the stalk in the PRE, RUN and END sections and shadows them (overwrites their presence) with other AFM's as instructed from the command line (such as when a developer compares his own AFM with another AFM, of a colleague, or from the PROD section of code).

This *hot-swap* facility allows developers to rapidly switch from one hypothesis to another, test single or multiple code pieces, and compare with reference new ideas in the group.

The "shadowing" convention work as follows: routines preceeded by *-s* (or simply nothing) are considered replacements in their respective sections (if a section is not mentioned, it is defaulted to *run*). Replacements are made until there are none more.

Addition of routines to one of the sections is also possible - this is performed with the *-a* qualifier. For example *iris pre.alfa1 beta1 -a ext.pre.delta* would mean hot-swap *alfa1* and immediately after add the list of functions following the *-a* qualifier (up to the next qualifier).

References

1. Dima M.O., The IRIS Development Platform and Proposed Object-Oriented Data Base // Journal of Software Engineering and Applications. – 2015. – Vol. 8. – P. 167-174.

2. ATLAS computing: Technical design report, ATLAS TDR--017, CERN-LHCC-2005-022. The Physics Analysis Tools project for the ATLAS experiment, Bruno Lenzi, ATL-SOFT-PROC-2009-006

3. Mato P. et al., GAUDI - LHCb Data Processing Applications Framework, Architecture Design Document, LHCb 98-064 COMP. - (1998).

РАСШИРЕННЫЕ СТРУКТУРЫ МОДУЛЕЙ ФРЕЙМВОРКА ДЛЯ ПЛАТФОРМЫ РАЗРАБОТКИ IRIS

Дима Михай-Октавиан, *д-р физ.-мат. наук, вед. науч. сотр.*
В.В. Кореньков, *д-р техн. наук, гл. науч. сотр.*
Объединенный институт ядерных исследований
(Россия, г. Дубна)

Аннотация. *Платформа разработки IRIS предоставляет услуги компиляции, межпроектной увязки и выборки данных, обеспечивая при этом возможность hot-swaps для сравнения производительности разрабатываемых модулей. В то время как предыдущая работа была посвящена самой структуре [1], в текущей статье подробно описывается структура модулей AFM, собранных в рамках IRIS. Разработка вызвана потребностью в расширенных модулях производительности, которые стандартизированы и независимы от фреймворков, концептуально связанных с C++98. В нем содержится основанное на примерах руководство для ученых по прямой реализации кода в отдельной модели компиляции шаблонной технологии C++11 (с видением соответствия Concepts^TS^, позволяющим относительно легко перейти на C++ 20).*

Ключевые слова*: платформа разработки программного обеспечения, C++11, технология общих объектов.*