Journal of Software Engineering and Applications, 2015, 8, 167-174
Published Online April 2015 in SciRes. http://www.scirp.org/journal/jsea
http://dx.doi.org/10.4236/jsea.2015.84017

The *IRIS* Development Platform and Proposed Object-Oriented Data Base

Mihai-Octavian Dima

Institute for Physics and Nuclear Engineering, Bucharest, Romania
Email: modima@nipne.ro

Received 14 January 2015; accepted 13 April 2015; published 14 April 2015

Abstract

Various code development platforms, such as the ATHENA Framework [1] of the ATLAS [2] experiment encounter lengthy compilation/linking times. To augment this situation, the *IRIS* Development Platform was built as a software development framework acting as compiler, cross-project linker and data fetcher, which allow hot-swaps in order to compare various versions of software under test. The flexibility fostered by IRIS allowed modular exchange of software libraries among developers, making it a powerful development tool. The *IRIS* platform used input data ROOT-ntuples [3]; however a new data model is sought, in line with the facilities offered by *IRIS*. The schematic of a possible new data structuring—as a user implemented object oriented data base, is presented.

Keywords

Software Development Platform, User-Defined Object Oriented Data-Base

1. Introduction

The *IRIS* platform originated as an improvement to the work under the ATHENA Framework [1] of the ATLAS [2] experiment. ATHENA is an Object-oriented/Multi-threading software developed at CERN for data-fetching and code running of ATLAS triggering, and reconstruction under a vast array of software contributions to all sub-systems. The platform was under development in 2004 and considerable overhead was encountered when accessing data and running over untuned sections of code pertaining to various sub-systems. The principal problem encountered was the slow code compilation and linking, a much faster framework being needed—with flexibility to allow exchange of software libraries among developers.

IRIS acts both as compiler and running environment, giving flexibility in comparing work of developers, routine hot-swapping and shared-object library creation. It was created to augment the development of LVL-2 trigger code and had as input data ROOT-tuple [3] skimmed data. The aim then was to be able to test 5 - 10 ideas/

hour, and promote a productive development environment. Envisaged was that work be flexibly hot-swapped in/out among developers using shared object libraries and that routines be "un/mounted" at will at any point ("bean-stalk" **Figure 1**).

Such a software development platform is desirable both for its reliability [4] and its applicability to a number of numerical developments [5].

The routines are "mounted" on the framework just like files are attached to i-nodes to populate a file-system (see **Figure 2**). This is in the main program, that basically lists the routines to be mounted—example below:

```
extern field* amix(int, char**)       ;
int main(int argc, char** argv)
{ field* ipx                        ;
  ipx = amix(argc, argv)              ;
  JOB dataqual(ipx, "job DATA QUAL" ,
                "pre.alfa1"     ,
                "pre.alfa2"     ,
                "run.beta1"     ,
                "run.beta2"     ,
                "end.gamma1"    ,
                "end.gamma2"    ,
            NULL)                   ;
  dataqual.run()                      ;
  delete ipx                         ;
  }
```

The *iris* executable will expect the presence in the running directory of *alfa*1.*cc*, *alfa*2.*cc*, *beta*1.*cc*, *beta*2.*cc*, *gamma*1.*cc*, and *gamma*2.*cc*. More "leafs" can be of course added to the iris stalk. After downloading and un-

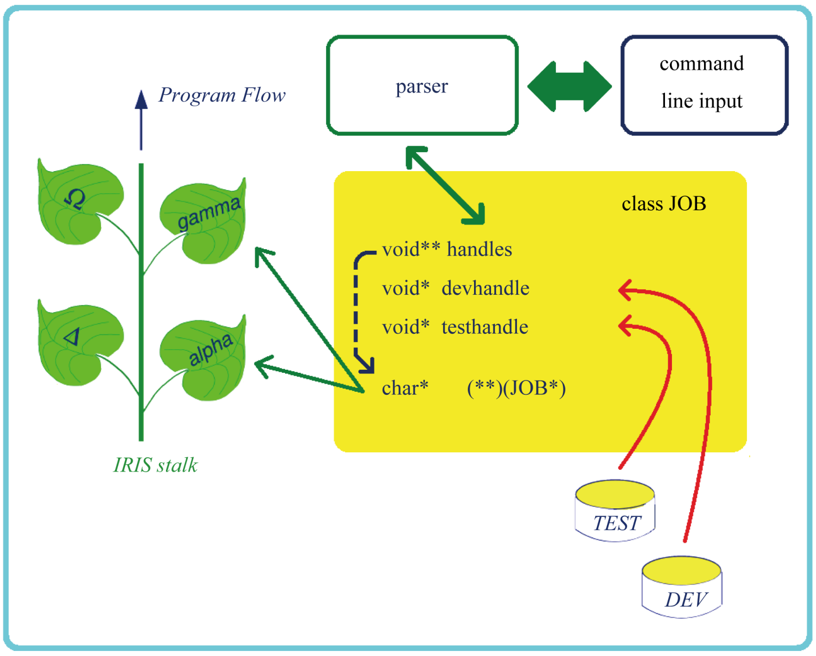

Figure 1. The IRIS platform relies on a mounter singleton defining handles (sockets) to functions found in shared object libraries. These are typed into char* (**)(JOB*) functions that actually are mounted in the exec-loop of the class. Various auxiliary functions are also performed, like compilation of routines that are under development—using the stdlib library. Shared objects are loaded with the dlfcn library.

packing the package, it can be compiled with the standard make. At this point *iris* will take control, compile the above files and make a *dev.so* shared-object library. Issuing the command *iris* will produce the following output:

```
Compiling alfa1  ... DONE ! (1 out of 6)
Compiling alfa2  ... DONE ! (2 out of 6)
Compiling beta1  ... DONE ! (3 out of 6)
Compiling beta2  ... DONE ! (4 out of 6)
Compiling gamma1 ... DONE ! (5 out of 6)
Compiling gamma2 ... DONE ! (6 out of 6)
----------
Building DEV shared lib ... DONE !
```

At this first stage it will not run the job. To run the job, issue: *iris beta*1. In full this means *iris -s dev.run.beta*1, or: shadow and replace the first function from the *RUN*-section with function *beta*1 (found in *dev.so*), and then run the job. The code will open *dev.so* for the default functions set in *t.cc*, and also the **.so* file in which the substitute function *beta*1 is found (also *dev.so*). It will build the necessary function pointers on the "stalk" and:

- run the *PRE*-section in the *JOB*-object constructor;
- in the *JOB.run*() function it will open the ROOT-tuple and run for each event all "leaves" in the *RUN*-section. The ROOT-tuple will close itself at the end of *JOB.run*() and finally
- the *END*-section "leaves" will be run in the destructor section of *JOB*.

2. Usage of *IRIS*

The iris executable is designed to give flexibility in development work: it is fast in data-fetching, fast in function-fetching [6] and versatile in compiling or just fetching functions (owned by the developer, or by other developers in the group). This 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. Below will be outlined the main use-cases of *iris*. The notation convention of iris is the following: fully qualified name refers to *file.section.function* and denotes that the user wants to place function *function* from file *file* in section section. Omitting the file defaults to *dev*, omitting the section defaults to *run*, and omitting the function defaults to ... a link-fault.

Compiling routines—most development work means having a fixed set of "environment"-routines that run everytime, and 1 - 2 routines that change as the developer improves the algorithm, or makes amendments to it. This requires the fixed set to be mentioned in t.cc as shown in the Introduction, and the amended set to be declared at run-time. First time *iris* is run (command *iris*), it produces the *dev.so* library in which the fixed environment will reside, together with the starting version of the routines to be amended. The developer implements some changes, and issues a command like *iris pre.alfa*1 *beta*1 *-c pre.delta omega*, where *delta* and *omega* are the new versions of *alfa*2 and *beta*2. (Renaming is not necessary, here it is done for code output illustrative purposes only.) This means: compile the new versions *delta* and *omega*, make the *test.so* library with them, shadow *alfa*1 with itself (no change) and likewise with beta1, then shadow *alfa*2 from *dev.so* with the new version from *test.so* and likewise for *beta*2. The output from *iris* will look like this:

```
Compiling delta ... DONE !
Compiling omega ... DONE !
----------
Building TEST shared lib ... DONE !
****************************
PRE jobs
---
ran >> alfa1
ran >> delta
----------------------------
RUN jobs
---
```

```
event nr. 0 ran >> beta1 >> omega ... DONE !
event nr. 1 ran >> beta1 >> omega ... DONE !
event nr. 2 ran >> beta1 >> omega ... DONE !
----------------------------
END jobs
---
ran >> gamma1
ran >> gamma2
****************************
```

Note that the code will print the function's name after it has been executed, which is useful for debugging purposes.

It is easy to see how the "shadowing" convention works: 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 (as defined in *t.cc*). In the above example only two replacements per section are possible. Addition of routines is possible, as it will be shown below.

If in addition to *-s* the *-c* is also present (as *-sc*, or simply *-c*), then *iris* expects the respective *files.cc* to be present in the running directory: it will compile them, build the *test.so* library and mount them, according to prescription, on the *iris* "stalk".

Swapping routines in/out—it is possible that the developer wants to check a reference library against developed code, or simply check a collegue's *dev.so* library which—for notation purposes—shall be called *ext.so*, the "external" library. This is one of the main strengths of *iris*, the flexibility to accommodate various contributions within the development group.

The same above operations would have been in this case: *iris pre.alfa*1 *beta*1 *-s ext.pre.delta ext.run.omega*. The package will skip compilation of any routines and simply load from the file *ext* the required functions to be

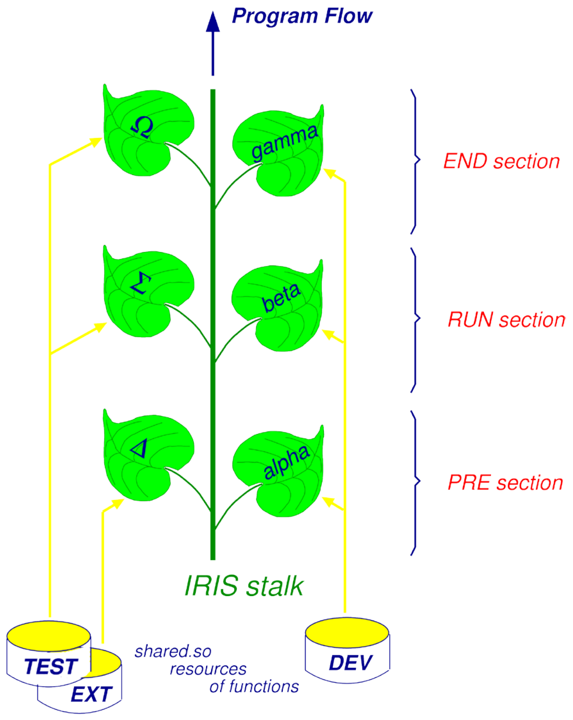

Figure 2. With iris work can be flexibly hot-swapped in/out among developers using shared object libraries—by "un/mount" at any point on a chain resembling a bean-stalk. The iris code compiles *dev.so* and *test.so* libraries and at runtime can open those for desired functions, or override them with functions from other *ext.so* libraries.

attached to the *iris*- “stalk”. The output in this case would be:

```
****************************
PRE jobs
---
ran >> alfa1
ran >> delta
----------------------------
RUN jobs
---
event nr. 0 ran >> beta1 >> omega ... DONE !
event nr. 1 ran >> beta1 >> omega ... DONE !
event nr. 2 ran >> beta1 >> omega ... DONE !
----------------------------
END jobs
---
ran >> gamma1
ran >> gamma2
****************************
```

It is possible to combine local function compilation with external libraries: *iris pre.alfa*1 *beta*1 -*s ext.pre.delta* -*c omega*, the output being:

```
Compiling omega ... DONE !
----------
Building TEST shared lib ... DONE !
****************************
PRE jobs
---
ran >> alfa1
ran >> delta
----------------------------
RUN jobs
---
event nr. 0 ran >> beta1 >> omega ... DONE !
event nr. 1 ran >> beta1 >> omega ... DONE !
event nr. 2 ran >> beta1 >> omega ... DONE !
----------------------------
END jobs
---
ran >> gamma1
ran >> gamma2
****************************
```

Adding routines—is performed by the -*a* qualifier. For example *iris pre.alfa*1 *beta*1 -*a ext.pre.delta* -*c omega* would mean hot-swap *alfa*1, immediately after add the list of functions following the -*a* qualifier (up to the next qualifier), compile *omega* and replace *beta*2 with it. The output would look like:

```
Compiling omega ... DONE !
----------
Building TEST shared lib ... DONE !
****************************
PRE jobs
```

```
---
ran >> alfa1
ran >> delta
ran >> alfa2
----------------------------
RUN jobs
---
event nr. 0 ran >> beta1 >> omega ... DONE !
event nr. 1 ran >> beta1 >> omega ... DONE !
event nr. 2 ran >> beta1 >> omega ... DONE !
----------------------------
END jobs
---
ran >> gamma1
ran >> gamma2
****************************
```

3. Persistent Object Data Base

IRIS relied on n-tuples as input data—in the form of ROOT-tuples [3]. This is flat-data and the current section offers a few ideas on how a similar, user designed, but object oriented data base could be written.

Work with C++ (F-90 and any algorithm language) has led mostly to relational data storage. Just like in the case of IRIS above, more was not considered needed. Starting with the early 2000's this began to change, persistent (non-flatened) object storage being more and more of need.

Writing data to permanent storage was formalized starting with the 1960's (relational data bases—RDB) and the 1980's (object oriented data bases—OODB).

The most widely accepted standard for OODB Management System (OODBMS [7]) is the ODMG 3.0 [8]. The ODMG group however is divided over options for a future 4th generation OODBMS standard. ODMG places 2 general sets of demands usually required by an OODBMS:

- DBMS conditions:
  1. persistence
  2. secondary storage management
  3. concurrency
  4. recovery
  5. ad hoc query facility
- OO conditions:
  1. handling complex objects
  2. accommodating object identity
  3. observing encapsulation
  4. handling types or classes
  5. complying with inheritance
  6. overriding combined with late binding
  7. extensibility
  8. having computational completeness

It became apparent in C++ evolution [9] that the objects not only are "constructed" by the user, but also the user should be responsible for their proper disposal (memory management, de-allocation). This can be extended: the user being also responsible for selecting and writing which data needs to go in a data-base (likewise in the destructor section of a class).

Of special attention here is the allocation of memory within objects "construction"—which the machine is not (and should not) be able to interfere with, the management of this resource being entirely up to the user. Such allocation may be virtual by inheritance (*i.e.*—"fish" allocates one size/(type) of memory, while "dog" allocates another, both inheriting "farm animal"). This led to the (correct!) prevalence of destructors and virtual destructors in C++ and the responsibility of the user in managing their memory de/allocation.

Naturally, this also led to a number of problems in large projects involving numerous contributors—that had little Quality Control (memory leaks, segmentation faults, etc.).

Taking a cue from this evolution writing persistent objects to permanent storage should have the same approach: namely making the user responsible for the proper writing of objects to the DB. Basically this proceeds in the same way that the objects are deleted in the destructor.

The model here proposed relies thus on the user, not the DB-software, or the operating system (OS) to realise these demands—just as the application relies on the user to properly write the destructor.

In principle the only memory accounting needed is that of the (new)-allocation of known C++ types. All arrays of user-defined objects need not be known in number, as each object's destructor is called by an iterative loop of the *delete*[] statement, by a (secret) counter held by the OS. Each destructor will know what to do (and what to write).

The same applies for deleting a generic class of objects—say "animal farm", the virtual destructor *delete*[ ]-ing (and writing to DB, relyant on each of its implementations) "cattle", "sheep", "chicken" ... accordingly.

The above become transparent with the use of smart pointers, though these are implementation dependent and perhaps better to be avoided.

Typical flatening of objects would be implemented by writing to 3 DB's (**Figure 3**):

- a *header-DB* (**HDB**): containing the definitions of the classes—indexed by order number
- an *object-DB* (**ODB**): containing flattened objects, indexed by type and order number. Order numbers would be flagged as \visible" and \invisible", depending upon the object being declared self-standing, or within a class—in 2 lists: v1, v2, v3, ... and i1, i2, i3 ... etc. This allows fast DB-querying, both internally and externally
- an *allocation-BD* (**ADB**): containing memory allocations of each object, indexed by order nr. and regarded as bit-streams (their deconvolution interpreted by the object-DB).

Relational keys—between the three are evidently needed; however they are fast, as the above indexing pertains little header volume (little impedance mismatch). A slim-tall example of relation key structure can be found in the JAZELLE-DBMS [10].

JAZELLE keys delete all horizontally related objects, although this is a requirement specific to algorithmic languages (C, C++, F-90, etc.). ADA for instance does not require DB fault-free compliance, as incomplete type declarations are common practice in this language, hence references to missing objects are tolerated.

Take the example "particle" in High Energy Physics, which can reference: "vertex", "track", "cluster", "mc-truth". Any of the objects may be missing, hence "particle" that references all, or themselves referencing among themselves, need not all disappear just because one is absent. Said absence causes a (solved or) unsolved DB-fault and is signalled by a nil-pointer. This feature is useful, as it saves time from loading un-necessary structures into the application, from the DB. This touches also on OO-8 above: "extensibility"—in the sense that

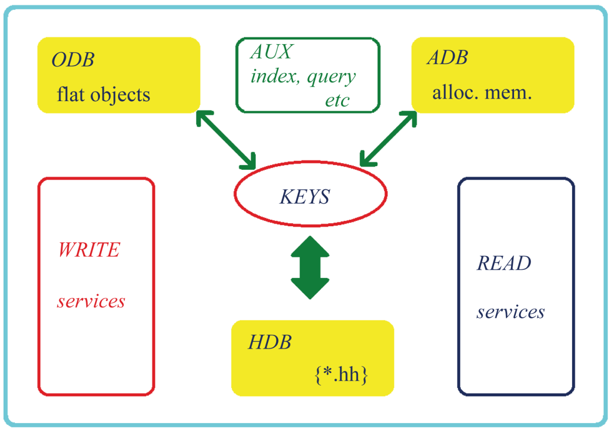

Figure 3. A section showing diagram of the proposed object oriented data base. The user (responsible for the objects' destructors) is also the implementer of the writing of objects. This is performed with the services offered by the DB, however in the structure format here shown.

it can be extended also with missing data types (having no specification), which however, may be added at a later stage, but, which do not prevent the partial-functioning of the DB.

As the user is the implementer of the read-constructor and write-destructor, this job will rely on DB "services" of accessing/storing data in the 3 sub-DB's.

Version-check services would also be provided issuing strong-warnings in case of outdate-mismatches (of the header files).

Transparent persistence of the DB is also fast, updating the ODB/ADB being nearly header free.

Acknowledgements

This work was supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI, grant number ANCS PN09370104. I am also thankful to the Wuppertal Physics Department for the 2004 stay and to the ATLAS Collaboration for access to the ATHENA Framework and Monte Carlo simulation data.

References

[1] ATLAS Collaboration—Duckeck, G., *et al*. (2005) Atlas Computing: Technical Design Report. CERN-LHCC-2005-022; Lenzi, B. (2009) The Physics Analysis Tools Project for the ATLAS Experiment. ATL-SOFT-PROC-2009-006.

[2] Aad, G., *et al*. (2008) The ATLAS Experiment at the CERN Large Hadron Collider. *Journal of Instrumentation*, **3**, S08003.

[3] Antcheva, I., *et al*. (2009) ROOT—A C++ Framework for Petabyte Data Storage, Statistical Analysis and Visualization. *Computer Physics Communications*, **180**, 2499-2512. http://dx.doi.org/10.1016/j.cpc.2009.08.005

[4] Adam, G. and Adam, S. (2003) Reliable Software in Computational Physics. *Romanian Reports in Physics*, **55**, 488.

[5] Adam, G., *et al*. (2006) Resolving Thin Boundary Layers in Numerical Quadrature. *Romanian Reports in Physics*, **58**, 155; Balaceanu, V. and Pavelescu, M. (2011) Neutronic Calculation System for CANDU Core Based on Transport Methods. *Romanian Reports in Physics*, **63**, 948; Necula, C. and Panaiotu, C. (2008) Application of Dynamic Programming to the Dating of a Loess-Paleosol Sequence. *Romanian Reports in Physics*, **60**, 157.

[6] Contrary to Popular Belief, Shared-Object Dynamic Binding Provides Faster Code through Memory "In-Page" Function Fitting and Considerably Faster Hot-Swap Times vs. Static Binding.

[7] O'Brien, J.A. and Marakas, G.M. (2009) Management Information Systems. McGraw-Hill/Irwin, New York; Atkinson, M., *et al*. (1992) The Object-Oriented Database Manifesto. In: *Building an Object-Oriented Database System*, Morgan Kaufmann Publishers Inc., San Francisco, 1-20.

[8] Cattell, R.G.G., *et al*., Eds. (2000) The Object Data Management Standard: ODMG 3.0. Morgan Kaufmann Publishers Inc., San Francisco.

[9] Bjarne Stroustrup (1989) The Evolution of C++: 1985-1989. *Computing Systems*, **2**, 191; Bjarne Stroustrup (1999) An Overview of the C++ Programming Language, in: The Handbook of Object Technology, Ed. Saba Zamir, CRC Press LLC, Boca Raton; Bjarne Stroustrup (1994) The Design and Evolution of C++, in: Addison-Wesley Publ.; Torsten Sehy (2012) *Evolution of C++*, Seminar on Languages for Scientific Computing. http://hpac.rwth-aachen.de/teaching/sem-lsc-12/EvolutionC++.pdf

[10] Johnson, T. (1990) JAZELLE Users Manual, SLAC-R-362.