The STAR Analysis Framework; Component Software in a Real-World Physics Experiment

C.Tull, W.Greiman, D.Olson, D.Prindle, H.Ward

Ernest Orlando Lawrence Berkeley National Laboratory, University of California
University of Washington, Nuclear Physics Laboratory GL-10 for the Solenoidal Tracker At RHIC Collaboration

The STAR Analysis Framework (STAF) is a highly modular framework written in C++ and designed around a CORBA-compliant software bus package. STAF provides a CORBA-compliant encapsulation of user-written data analysis algorithms written in FORTRAN, C, and C++ which allows the seamless integration of physics software components and system-like software components controlled at run time by a high-level scripting language and/or by graphical user interfaces. We present in this paper our experience with the STAR Analysis Framework in particular, and with the software-bus/component-software approach to software engineering in a physics experiment in general. We also discuss our choices of communication and interface protocols and the importance of designing these protocol layers in a manner which allows for "graceful retirement" of said protocols as a means of prolonging the effective lifetime of software systems.

Key words: STAR, RHIC, object-oriented, data analysis framework, CORBA, C++, FORTRAN

1 Introduction

STAR - Solonoidal Track At RHIC

The Solenoidal Tracker At RHIC (STAR[1]) collaboration is a large international collaboration of about 400 high-energy and nuclear physicists located at 37 institutions in Brazil, Croatia, Germany, Israel, Poland, Russia, and the United States. When the Relativistic Heavy-Ion Collider (RHIC[2]) at

[1] http://www.rhic.bnl.gov/STAR/
[2] http://www.rhic.bnl.gov/

Brookhaven National Laboratory (BNL), Upton, NY, USA comes on line in 1999, STAR will begin a period of data taking and concurrent data analysis that is expected to last until at least the year 2010.

The Software Challenge

During normal operation, the STAR collaboration expects a data acquisition rate of approximately 250 Tbytes of raw data per year for the projected ten-year operating lifetime of RHIC. Factoring in Data Summary Tapes (DSTs) derived from the raw data and anticipating comparable statistics from Monte Carlo simulations yields a large fraction of a Pbyte per year that STAR must manage, process, and analyse.

As is typical for this early stage in the lifecycle of a large physics collaboration, almost all of the currently extant analysis software is presently undergoing intense development. Although much of the reconstruction software will approach an asymptote of finality early in the production stage of analysis (i.e. when a steady-state is achieved between data acquisition and data analysis rates), much of the physics software will continue to change significantly over the entire duration of the experiment and data analysis. During this same time period, the physicists in charge of various aspects of the physics analysis software will also continuously change as graduate students, post-doctoral researchers, and senior scientists join and leave the collaboration or change their focus on the analysis and physics.

This fluid nature of analysis software is fundamental to a dynamic physics analysis process. However, without careful planning a great deal of programming overhead can be required to maintain a complex analysis system for a prolonged time period.

Software Infrastructure

Much of the software needed in the analysis of experimental data is not strictly related to the detectors used nor to the physics being investigated. Hence, though crucial to a successful analysis of the data, this software infrastructure has often been relegated to a much lower priority than the physics algorithms comprising the data analysis. Though the primacy of the physics software in a physics analysis is indisputable, this software infrastructure gains in importance as an experimental collaboration grows larger and lasts longer.

In STAR, the SOFtware Infrastructure (SOFI[3]) group is charged with the responsibility of developing and maintaining general purpose tools and software for use by physicists in the analysis of data and investigation of theory and results. The STAR Analysis Framework (STAF) is a major part of the software solution that has been, and is being developed by the SOFI group for STAR.

[3] http://www.rhic.bnl.gov/STAR/html/sofi_l/sofi.html

2 Software Architecture

In a collaboration with many writers and users of analysis code, some mechanisms for co-ordination of and communication between analysis software elements must exist to facilitate an efficient overall analysis process.
One way of doing this is to write each analysis element as a stand-alone program executing one step in the analysis process and communicating with other analysis elements (other programs) through some well-defined file formats. The chain of analysis is then normally a batch job which executes the programs in order, cleaning up temporary files and saving results to permanent files.
Though this approach has been used in many experiments, it works best when a very limited number of people are responsible for the entire analysis. This allows changes to data formats or data flow to be made with the reasonable expectation that all appropriate programs will be updated in a synchronised fashion. One drawback with this approach is that it lends itself to duplication of effort and/or code. Each analysis program must handle its own I/O, memory management, etc. This drawback can be mitigated with a set of common utility functions in a central library, but does not lend itself easily to global changes in data format, or in the analysis chain.

Framework vs. Shell

Another approach which has been successful is the use of an analysis shell. An analysis shell is a generic program which handles the system-like functions of data analysis such as data I/O, memory management, flow control, etc. without doing any real analysis of data. The actual data analysis is done by "analysis modules" conforming to some API (Application Programming Interface) which allows the modules to be plugged into the analysis shell in a modular fashion. The analysis shell invokes each analysis module within the analysis chain, either passing data to the module via the API, or presenting data when requested by shell functions invoked within the module. Often the analysis shell also contains other general purpose tools for investigation and analysis of data (e.g. histogramming, plotting, sorting, simple calculations).
We have extended this concept of plug-in modules from the user-written analysis code to the central, system-like analysis shell. By dividing the functions of the analysis shell into "domains" and adopting an interface standard for these system-like functions, we provide a customisable framework for data analysis.

Graceful Retirement

STAR begins taking data in 2 1/2 years, and will continue to take data for at least ten years. This means that short of completely replacing the analysis system with another system part way through the experiment, the software design must have a lifespan of $\sim$15 years. This is an incredibly long time in the dynamic world of computer software and hardware. To put this into perspective, one need only consider the state of computing in physics 15 years ago. In 1982, the hot new machine in physics was the VAX 780, FORTRAN 77 was a newer language than FORTRAN 90 is today, and your choice of color terminals was green or amber.

We conclude that it is unrealistic to expect any software system written today to survive unmodified for 15 years. Hence, any sensible design for a software system needed to last that long must incorporate, at a fundamental level, the concept of graceful retirement (i.e. replacement in a controllable manner) of any and all of its constituent components.

Consider further that STAR encompasses many distinct physics programs, each with its own unique analysis needs and that scores, perhaps hundreds, of physicists of diverse backgrounds and skills will be using and contributing to the analysis system over its lifetime, and the magnitude of the challenge begins to be appreciated. Our approach to the STAR Analysis Framework addresses each of these challenges in a manner which we believe can be expected to succeed over the long term as well as the short.

By making the division between the framework kernal (i.e. the software bus) and the plug-in service packages (see below) clean and well defined (what we have termed vertical modularity), we provide for the graceful retirement of communication protocols and interface standards. By dividing the system-like services into autonomous packages (horizontal modularity), we allow graceful retirement of code libraries, as well as easing the burden of code maintenance.

Finally, rather than try to define all of our own interface standards, external data representations, etc., we have tried wherever possible to adopt open software standards from the computer industry and computer engineering communities. These standards are often better designed and supported than home-grown standards due to the hundreds or even thousands of man-hours devoted to their development. They are generally well documented, providing guidance for programmers attempting to collaborate over long distances. Some of these standards allow use of powerful commercial software. And even standards which don't survive over the long term often provide migration paths and/or tools to other, functionally equivalent standards.

All this means that there are many advantages to adopting a framework architecture, both from the users' perspective and from the perspective of the framework programmer.

3 STAR Analysis Framework

Vocabulary

A few terms need to be defined at this point to facilitate discussion of the STAR Analysis Framework. Each of these terms is examined in detail later in this article.

- Table
  A table is the primary container of data for STAF. Each table instance has a name and a type and is composed of rows and columns. Each table column has a name and type. Column types are scalars, vectors, arrays, or structures of the eight CORBA primative variable types (char, octet, short, long, unsigned short, unsigned long, float, double). A table row is a fixed-length record of one data cell for each column. Tables can be thought of as arrays of C structs or FORTRAN structures. A table type is a named discriptor of the table column types and names (e.g. `"struct keyPointlong key; float x,y,z;;")`.
- Dataset
  A dataset is a container for tables and/or other datasets. Since datasets and tables can be referenced by links, datasets form a general directed graph. However, STAF generally limits datasets to be directed acyclic graphs or hierarchical trees.
- Physics Analysis Module
  A Physics Analysis Module (PAM) is a user-written function in C, C++, or FORTRAN which adhears to one of the STAF PAM APIs. Currently there is only one API for PAMs in STAF. A PAM which adhears to this API is called a type 1 PAM. A type 1 PAM returns a STAF condition value and takes 2*R arguments, where R is the rank of the PAM (i.e. the number of data tables upon which the module operates). An analysis module processes only data tables of predefined types or compatable types.
- Analysis Service Package
  An Analysis Service Package (ASP) is a software package which serves a system-like function of the analysis framework. An ASP may depend explicitly upon other ASPs (for instance, an ASP for data I/O would depend upon the ASP responsible for memory management) and operates independently of all other ASPs. In general, an ASP is not associated with particular data table types.

STAF Architecture

A block diagram of one possible configuration of STAF is shown in Figure 1. In this configuration, five ASPs and nine PAMs are loaded. All of the ASPs communicate with other ASPs and with the software bus (SOC) through CORBA compliant interfaces (shown as hashed bars). Each analysis module interfaces through the Type 1 PAM API (shown as dotted bars) to a C++ object which presents a CORBA compliant interface to the rest of the system. The diagram illustrates the independence of the SPX and TDM packages on all other ASPs, and shows that the DIO, TOP, and AMI packages depend upon the TDM package (i.e. TDM must be loaded before DIO, TOP, or AMI can be loaded or used. The other interfaces shown (between TDM and DSL and below) are internal interfaces which are neither CORBA compliant nor Type 1 PAM APIs, demonstrating that virtually any pre-existing software package can be adopted to the STAF framework by encapsulation in a CORBA compliant set of C++ wrapper classes.

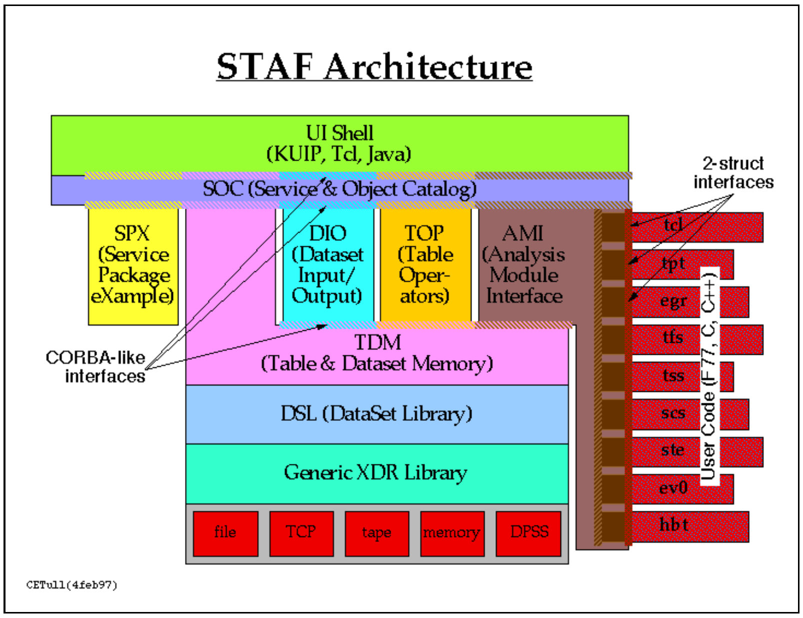

Fig. 1. Block diagram of the STAF architecture for one possible configuration of ASPs and PAMs.

Software Bus

Simply stated, a software bus is a pathway of communication between software components. It provides a common centralised resource for locating software components and routing communication and control through well regulated channels.
Our choice of CORBA as a software bus standard was dictated by the distributed, heterogeneous nature of our computer resources, our reliance on UNIX, and by our investigations into the availability and suitability of open standards two years ago when the choice was made. Other choices are now available (DCOM/Active X, distributed Java), but CORBA still remains the most viable choice for the immediate future.

4 CORBA

The Common Object Request Broker Architecture (CORBA) standard is a non-proprietary software bus specification developed and maintained by the Object Management Group (OMG[4]) - a consortium of over 700 computer industry hardware and software companies. CORBA can be considered an object-oriented client-server system which offers location transparency and access transparency for objects within the systems. This allows programmers to write code without knowledge of or concern about where needed software

[4] http://www.omg.org/

resources exist in the application. The ORB makes sure that objects communicate properly whether they are in the same process, in different processes on the same machine, or on different machines connected by a network.
Technically speaking, the software bus in STAF is implemented as an ASP upon which all other ASPs in STAF depend. The Service and Object Catalog (SOC) package contains two pure-virtual base classes (socObject and socFactory), and one special class (socCatalog) of which there can be only one instance in any STAF process (This is referred to as a single-instance class.). In a collocated STAF process, the socCatalog object mimics the static invocation interface of the ORB in CORBA, locating requested objects and negotiating communications between objects within the system. Dynamic binding to objects is accomplished through dynamically linking to shared libraries. When linked with a commercial CORBA implementation (such as Orbix from Iona Technologies), the socCatalog object acts as a gateway between STAF objects and the real ORB. By inheritance from the socObject base class, every object instantiated within the system is automatically registered with the socCatalog object at creation. Each object in the system can be identified and located by its type (class) and name (instance) or by a unique identification number (the idReference) from C++, C, or FORTRAN through queries to the socCatalog object.
Taking a cue from thin client architectures, the normal operational mode for STAF is as a "thin executable" which contains only the user interface shell, the SOC package, and error handling. All other functionality is dynamically loaded as needed. Thus users of the system never need to build new executables, just new object libraries. Incidentally, this obviates the need to compile different executables for interactive and batch processes. The default STAF executable is already as stripped down as possible. A batch process simply does not load ASPs whose functionality are not needed.

Analysis Service Packages

Each Analysis Service Package in STAF is implemented by a C++ class library containing exactly one object factory class and zero or more worker classes. The object factory class is a single-point contact for the outside world to members of the ASP. An object factory has only three categories of functions: creating objects, finding objects, and deleting objects. In general, every object in STAF is locatable through either the socCatalog or through the object factory which created it.
When an ASP is loaded, two functions are called. The first function (the *_init() function) initialises everything necessary for the ASP to start-up. This may include loading any other ASPs upon which this ASP depends, dynamically linking to necessary functions, and/or executing initialisation code for a software library. The second function (the *_start() function) starts the ASP by loading the user interface and instantiating the object factory. Hence, anytime an ASP is loaded, its object factory is available to the system.

Object factories are examples of what are termed "few-instance classes"; meaning that no more than one instance of an object factory class exists in any process. However, in a distributed environment, several instances of the same object factory class can exist in separate, distributed processes.

ASP	Description	classes
AMI	Analysis Module Invoker	1
DIO	Dataset Input/Output	5
DUI	Dataset Unix-like Interface	0
EML	Error Messaging & Logging	0
LEV	Logging of Environment & Versions	0
NIO	Ntuple Input/Output	1
SOC	Service & Object Catalog	0
SPX	Service Package eXample	2
TBR	motif Table BRowser	1
TDM	Table & Dataset Memory	3
TNT	Table to NTuple	1
TOP	Table OPerators	4

Table 1: List of available ASPs as of 17feb97.

In addition to the object factory class, most ASPs contain other classes which perform the actual work for the ASP. For example the Dataset Input/Output package (DIO) contains the object factory class (dioFactory), a generic stream class (dioStream) and several specific stream classes (dioFileStream, dioTapeStream, dioSocketStream) which are subclasses of dioStream and handle specific kinds of I/O streams. Obviously, the attributes, methods, and implementation of these classes is highly dependent on the work performed by the ASP. With few exceptions, these worker classes are "many-instance classes"; meaning that the number of instances of each class in the system is totally arbitrary.
Table 1 shows a list of the ASPs currently (17feb97) available in STAF.

Physics Analysis Modules

Of course, Physics Analysis Modules (PAMs) are the raison d'tre for STAF. Paradoxically, the requirements for the interface to PAMs is that it ought to be 1) powerful and 2) easy to use. Powerful, since almost all of the complex, computationally intensive code in the system exists behind the analysis module API, and it would be absurd to choke off the analysis code from the full richness of the data upon which it operates. Easy to use, because the largest number of writers of code will be interfacing to STAF through the PAM interface, many of whom will not be interested in a steep learning curve to integrate their simple analysis code into the system.

We have developed two, complimentary approaches in STAF to accommodate this seeming duality of interface complexity. The first is the definition of several types of PAM interfaces, of which type 1 is the first and only interface thus far defined. The second approach is to provide tools (code generators and compilers) which take some of the work out of the process of interfacing user code to the analysis framework.

5 IDL & stic

The Interface Definition Language (IDL) is a purely descriptive language (i.e. no operators) from the OMG used to define interfaces to CORBA objects. It is similar to C++ in syntax and variable name scoping and is a well accepted industry standard. STAR uses IDL to define table types, define analysis module interfaces, and to define service package interfaces.
The tool used for interfacing analysis modules in STAF is the STAR IDL Compiler (stic). Stic processes data table type definitions and analysis module interface definitions in IDL and produces C header files, FORTRAN include files, and glue code which, when executed in the STAF environment, instantiates an Analysis Module Invoker object (amiModule) which, in turn, provides a CORBA-compliant C++ wrapper for the analysis module. Stic also generates template files in both C and FORTRAN for the analysis module, making it quite easy to get started, even if the user is totally unfamiliar with the STAF programming API.
The amiModule object for a particular PAM, knows the rank of the analysis module, and the table types and I/O mode of all the table arguments to the analysis module. Hence, the amiModule object can perform run-time type checking on data tables before invoking the user-written function. But the main purpose of the amiModule object is to make the user code callable from the user interface. Invoking the call() method for an amiModule object from the command line with a sequence of tables as arguments, will cause the amiModule to pass the data associated with each of those tables to the user-written analysis module through the type 1 API.

6 Type 1 API

A type 1 analysis module is a simple C, C++, or FORTRAN function returning a 4 byte STAF condition value indicating either success or failure. The function has 2*R arguments, where R (rank) is the number of data tables the analysis module processes (reads and/or fills) on each call. The two arguments for a table are 1) a one-row fixed format meta-data struct describing the table, and 2) a pointer to the contiguous block of memory containing the table data, accessable as an array of structs.
Inside the analysis module the data are accessed as normal program variables. No special code is necessary unless table memory is to be reallocated. This

permits testing of the module code outside of the STAF environment and greatly simplifies the integration of legacy code into the analysis chain.

Dynamic Memory Allocation

Simple functions exist for dynamically allocating table memory in C or FORTRAN. For both languages, the API is the same. The user passes the metadata struct and the table data struct, and specifies the number of rows of the table desired. STAF will malloc the proper amount of memory, copy the table data from the old memory address to the new, and free the old memory.
For dynamic memory allocation in FORTRAN analysis modules, one extra step is required. When processing the PAM IDL file with stic, the "-dynamic" argument must be specified.

User Interface

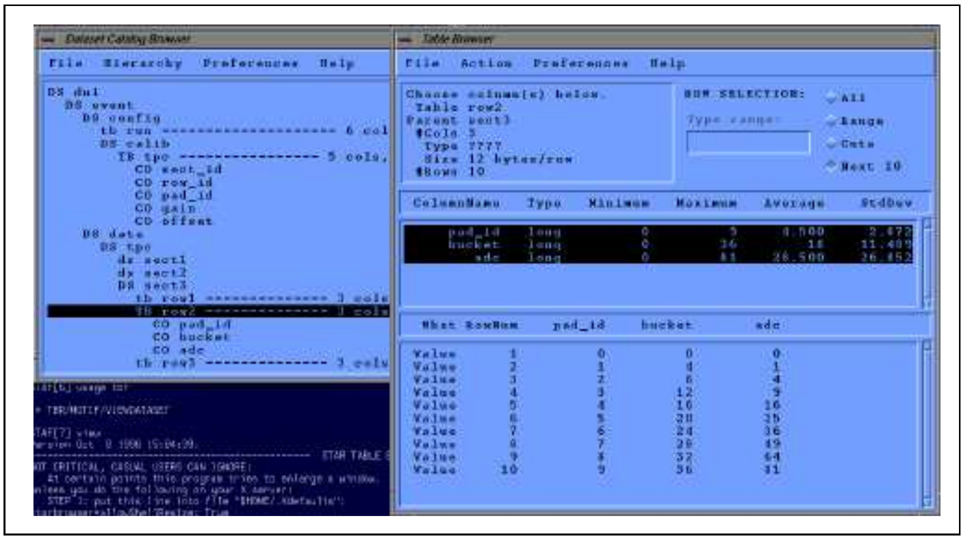

Fig. 2. Motif table browser GUI.

Graphical User Interfaces (GUIs) are impressive in software demonstrations and can help novice users learn a software package quickly and easily. However, a scripting interface usually provides more power and control with less work for the experienced user. In STAF, the primary user interface is considered to be the scripting interface. Any GUI interfaces are layered on top of, and consequently should reflect, this main scripting interface (e.g. see Figure 2).
Furthermore, the scripting interface is directly determined by the underlying object interfaces. Each object attribute and method function in an IDL file translates into a single command in the scripting language. This provides users of STAF with complete control of the system and the ability to program the analysis framework in a way that is directly translatable to actual code.
This run-time programability eliminates the need for a static, compiled event loop in all but the highest cycle applications. Provided the scripting language used has a sufficient level of flow control and arithmatic built in (as do most

scripting languages in use today), users can write and test very complex event loops in short order.
Once again returning to the litany of graceful retirement, this approach also means that when one scripting language is replaced with another, the look and feel of the user interface needn't change dramatically. Consider the following three examples of opening and reading a file in C++, KUIP, and Tcl.

- C++:
```
dioFileStream *oldat = dio→newFileStream("oldat","old_data.xdf","R");
oldat→getEvent(".");
```
- KUIP
```
DIO/NEWFILESTREAM oldat old_data.xdf R
GETEVENT oldat .
```
- Tcl
```
dio newFileStream oldat old_data.xdf R
oldat getEvent .
```

Although the syntax of the three languages is different, requiring the KUIP and Tcl commands to mirror the underlying C++ code makes the transition between any of the three options straight forward. In fact, in many cases the translation between two scripting languages, or between a scripting language and the associated C++ code can be done with a translation program.

7 Conclusion

Due to the STAR collaboration's size and geographical distribution, its data volume, and its long lifetime, STAR faces many of the same software and computer challenges that the next generation of high-energy physics experiments will be facing at facilities such as CERN's LHC in the next millenium. Consequently, our experience should help inform the high energy physics community's thinking on software architectures for the 21st century.
STAF has been in widespread use within STAR since June of 1996 for a variety of simulation and analysis tasks. And at present there are approximately 30 STAR physicsts actively developing software within the STAF framework. The Time Projection Chamber (TPC) system tests being conducted at the Ernest Orlando Lawrence Berkeley National Laboratory (Berkeley Lab) is making especially heavy use of the network I/O features of STAF by writing data to the network using the XDR-based Dataset Format (XDF), part of the Dataset I/O component of STAF.
We have found that adopting a software bus/component software architecture for STAF very fruitful on many fronts. The architecture greatly facilitates software development, changes, and porting, and produces code which is more easily testable, and consequently more robust.