ATL-DAQ-PROC-2015-012
15 May 2015

A Validation System for the Complex Event Processing Directives of the ATLAS Shifter Assistant Tool

A Santos[1], G Anders[2], G Avolio[2], A Kazarov[3], G Lehmann Miotto[2], I Soloviev[4]

[1]Universidad Nacional de La Plata (AR).
[2]CERN.
[3]B.P. Konstantinov Petersburg Nuclear Physics Institute - PNPI (RU).
[4]University of California Irvine (US).

E-mail: `Alejandro.Santos@cern.ch`

Abstract. Complex Event Processing (CEP) is a methodology that combines data from many sources in order to identify events or patterns that need particular attention. It has gained a lot of momentum in the computing world in the past few years and is used in ATLAS to continuously monitor the behaviour of the data acquisition system, to trigger corrective actions and to guide the experiment's operators. This technology is very powerful, if experts regularly insert and update their knowledge about the system's behaviour into the CEP engine. Nevertheless, writing or modifying CEP rules is not trivial since the used programming paradigm is quite different with respect to what developers are normally familiar with. In order to help experts verify that the rules work as expected, we have thus developed a complete testing and validation environment. This system consists of three main parts: the first is the data reader from existing storage of all relevant data streams that are produced during data taking, the second is a playback tool that allows to re-inject data of specific data taking sessions from the past into the CEP engine, and the third is a reporting tool that shows the output that the rules loaded into the engine would have produced in the live system. In this paper we describe the design and implementation of this validation system, highlight its strengths and shortcomings and indicate how such a system could be reused in similar projects.

1. Introduction

The ATLAS experiment [1] at CERN acquires physics data via a complex data selection and acquisition system consisting of software processes running on about 20000 cores interconnected via high performance Ethernet networks. Any member of the ATLAS collaboration (over 3000 scientific authors) can register as an operator, after having attended general as well as dedicated training sessions and having fulfilled a minimum of two shifts *shadowing* an already experienced operator ("*shifter*"). Consequently, and while there is always an on-call expert available, the control room is now populated by around 8 "shifters" who are in charge of different aspects of the experiment and not always have an in-depth knowledge of the components they monitor.

In order to retain the excellent data taking efficiency of the ATLAS experiment, the TDAQ [2] group has pushed developments in two directions: on one hand as many procedures and recoveries as possible were automatized, on the other hand a tool called the Shifter Assistant [3] (SA) was created.

The SA is a tool which assists the shifter in his/her daily work, diagnosing problematic situations and assisting in problem solving, with some special indications on which steps to follow. It works by providing checks and evaluating conditions in real time, processing, analyzing, and correlating heterogeneous sources of information. Once a condition is met, it propagates the appropriate message to the shifter at the control room or, for instance, sends a SMS or an e-mail to the relevant expert on-call expert. The SA has encountered a great success and has extended its initial scope from the data acquisition/selection area also to detector functioning and data quality. It is based on a Complex Event Processing (CEP) engine called Esper [4] and its knowledge base (under the form of directives) is coded using an Event Processing Language (EPL) modeled on SQL.

With the expansion of the knowledge base the SA was faced with the usual problem among expert-like systems: the management of the knowledge base, in terms of extensibility and testability. Since the SA was developed during a data taking phase (2010-2012) it was decided to adopt a very centralized approach, with directives being coded and tested by the SA development team, on request from different experts. During the LHC's Long Shutdown I period (2013-2014), it was decided to create a service which would allow different experts to write, test and commit their own directives to the running environment, without depending on the availability of the SA development team.

Testing is an integral part of software development and the sole mechanism that can ensure good quality in terms of functionality and performance. While unit testing is supported in most environments, a complex system cannot be put in place without extensive integration testing.

A service was developed to test the SA directives by executing the system inside a sandbox, using a configurable replica of past and archived data, events and ATLAS applications configuration. This approach allows the validation of directives in a controlled and independent environment, keeping a history of past tests. It is based on a software stack using an Oracle database, P-BEAST [5], C++, and Java at the back-end, and Python [6] and Django [7] at the web user interface front-end.

The SA testing procedure is called the *Shifter Assistant Replay Testing Model*, or *SAReplay*.

2. Testing of SA directives

The SA is an application with a modular design, and both the source of events and the alerting system can be extended or updated at runtime or configuration. It consumes different sources of events: IS [8], a service that allows sharing information between the ATLAS applications; ERS [9], a service that offers a centralized way of keeping track of messages produced by ATLAS applications; and OKS configuration [10], which is the ATLAS configuration service, a set of utilities for handling XML files defining the configuration of the ATLAS applications.

SA directives are encoded as XML files containing a specific data structure defining the EPL statements required by the Esper engine, the context and human readable messages for the alerting system, and the delivery method of the alert. All the generated alerts are stored in a database and made available in the ATLAS control room through a dedicated web application. Nevertheless, the SA also supports sending emails and SMS messages, writing the alert message to a file in the filesystem, or creating a new ERS message.

While CEP is a powerful tool, writing correct EPL statements is not trivial, particularly for newcomers to the CEP paradigm.

Directive developers should pay special attention to the amount of alerts produced, and their frequency. Having a large number of alerts would effectively kill the purpose of the system, since it may become very difficult for the person receiving the alert messages to keep up with them; other alerts from other directives may get lost in the produced noise.

For these reasons there was the need to provide a complete SA testing environment with a tool allowing to test directives in an environment as close as possible to the production system. The testing environment should provide the functionality to verify the syntactic and semantic validation of the directives with a clear user interface to fetch error messages and their context of occurence.

For testing the SA directives a black-box functional testing approach was chosen. The engine is started but, instead of being linked to the real ATLAS infrastructure, fake event injectors and alert processors are put in place in the engine, effectively allowing to run and test all directives in a controlled and replicable environment.

One shortcoming of the chosen testing approach is the selectivity of the data source to be read. Before starting a new testing execution, developers need to choose a subset of archived data to read. A bad choice of the data-set may cause the test to be incomplete and inaccurate. For this reason, developers are encouraged to first test their directives on a minimal data set for performance reasons before running the directives on a larger data set. If the outcome of the execution is satisfactory, a rerun of all directives on a larger data set should be performed.

3. The Replay Mode

3.1. Testing overview

Testing the SA directives is a procedure that can be easily followed, the *SAReplay* model. It begins with the definition of the EPL statement, then the definition of SA directive to be tested, and finally the SA engine is started with mocked input and output processors. Internal documentation is provided detailing specific modules, interfaces, helper functions and specialized event structures available within the SA. The documentation also offers common use cases which can be adapted to specific conditions to be tested.

SA directives are created and placed inside an XML file. Once created, a time interval to test the directive has to be selected: the developer may either input the concrete time interval or select a Run Number, which represents a specific interval of time for the running of the ATLAS applications infrastructure.

The testing of a SA directive is done by taking archived data and initializing the SA and the Esper engine with a different data source than the normal system in production. Instead of taking events, data and configuration from the ATLAS infrastructure, the *SAReplay* module initializes the readers and injectors from ATLAS archived data sources. The sources are replaced with a fake event injector, effectively isolating the SA from the ATLAS production infrastructure.

By these means, the *SAReplay* mechanism is the set of procedures required to read and sort the interesting and archived data from storage.

Once a time interval is selected, the SA engine is started, the fake events injector is put in place, all the streams are sorted and each event is recreated and re-injected back on to the Esper queue of events one at a time. Since the SA alerts may produce external effects, the Replay mode of the SA redirects all of the alerts to a file effectively mocking the alerting system.

3.2. Reading archived IS data from P-BEAST

Creating a fake injector of events for the SA required the creation of readers for the three sources of archived data.

For IS objects, the P-BEAST service is available: it stores IS data and provides C++ and Java APIs for reading values from archives. This interface was created specifically for the data storage and access patterns required for IS.

Given the complex data access patterns required by the P-BEAST interfaces, a new tool, OSIRIS, was developed in order to fully exploit the P-BEAST's potential, to abstract the low-level details of reading mixed data from it, and to efficiently allow the construction of a single timeline with all the history of events stored in the archives. This tool makes the required calls for each attribute of each object to be retrieved, sorts each time series of events, and produces a unified format for constructing the IS objects to be inserted as events to the SA.

From an efficiency point of view, and since interesting executions of the Replay mechanism may require reading several gigabytes of data, the tool needs to read data from archives in a smart way. While the P-BEAST API allows to read several hours of data at once, all data is stored in RAM and, for reading large amounts of data computers with vast amounts of memory would be needed. For this reason, when the OSIRIS tool is requested to read long periods of time, it reads data in shorter intervals and then joins all events together, allowing to read large amounts of data (approximately millions of events and several gigabytes) using a limited amout of memory.

3.3. OSIRIS API

OSIRIS is divided into three layers. The first and lower layer is an abstraction on top of the P-BEAST C++ interfaces. P-BEAST provides two independent interfaces to read archived data. The first and historically original P-BEAST interface is the EOS [11] interface, which allows to read P-BEAST data directly from disk storage. EOS is the Exabyte Storage Service at CERN, a service to efficiently store and retrieve large amounts of data on millions of files. The second P-BEAST interface is built as a CORBA [12, 5] interface, and reading P-BEAST data requires the communication with the P-BEAST servers over the network.

While both interfaces offer the same functionality, the actual P-BEAST API calls required to access the data are different. Consequently, the first layer of abstraction in the OSIRIS tool is the abstraction of the source of the P-BEAST data, so that data can be transparently read from any source. The low-level abstraction layer offers access to both the meta-data of the archive and the data itself, by using two different subclasses of one single C++ abstract class, following the well-known object-oriented design pattern strategy [13].

The second layer of abstraction of OSIRIS uses its low-level interface to read P-BEAST data in a smart way. This layer requests P-BEAST meta-data, builds the list of operations required to make to the data readers, and one at a time performs individual reads to the archives. Each read operation returns a new time-series of values represented as an array of data-points where all the values share the same data type, *i.e.* string or integer. From a P-BEAST point of view, a time-series is an array of values, each with its own range of timestamps, where the range represents the interval of time where the attribute had that specific value.

Since the Replay mechanism needs the creation of a single sorted timeline, another abstraction to mix time-series of different data types was required. The first option considered was to find a common representation of each value, for example a string, and construct a single array with all the elements having this common data-type. However, after some initial tests it was found that this approach was not efficient, not only from a run-time point of view, but also from a memory usage point of view. Representing every value in memory as a string imposes a heavy overhead on the memory use, especially in the case of millions of values whose native representation would only be a few bytes of memory.

After careful consideration of different representation alternatives, the final approach chosen to represent the single timeline of events was a sorted list of time-series, creating an array of arrays. Each individual array represents a time-series consisting of elements of the same data type, and this array is wrapped into a type erasure object hiding the underlying data type (different time-series may contain data of different data type). Since the values of each time-series are already sorted, the creation of a sorted array of time series can be done by sorting the sequence by the first element of each array.

The first element of the sorted sequence is the first element in the first array. To advance to the next element of the general sequence, this first element needs to be removed from its corresponding array, and the array of arrays should be resorted again. However, from

Figure 1. Algorithm to build a single timeline of time-series.

```
Data: The list L of time-series
Result: A single timeline of events
1  t ← new array;
2  for $x \in L$ do
3      e ← wrap(x) ;
4      add e to t;
5  make_heap(t);
6  while t is not empty do
7      s ← t[0];
8      e ← current(s);
9      process(e);
10     advance(s);
11     pop_heap(t);
12     if not end(s) then
13         push_heap(t);
14     else
15         remove last element of t;
```

Figure 2. Example OSIRIS C++ API usage.

```
osiris::api* api = new osiris::api_corba(...);
std::vector<std::string> object_names;
api->list_objects("ATLAS", "aDataType", "anAttribute", object_names);
```

a performance point of view, there are two expensive operations being done here: the first expensive operation is removing an element from an array, and the second expensive operation is to re-sort an array of elements when only one element was updated.

To solve the first expensive operation, each array can be wrapped in an abstract object with a set of operations to operate over the time-series and, by keeping a pointer to the "current" element of each time-series, the deletion of the first element of the array operation can be replaced by the advance of the pointer by one. When the pointer reaches past the last element, the time-series is consumed and can be removed from the general sequence.

To solve the second expensive operation, the sorting of the array of arrays operation can be replaced with a min-binary-heap, a data structure that efficiently allows to insert a new element and get the minimum element from it. By keeping a min-binary-heap of time-series the efficient construction of a general sequence of sorted values can be realized. For this implementation, the C++ STL functions used were: `std::make_heap`, `std::push_heap`, `std::pop_heap` [14].

The definition of the proposed algorithm can be found in Figure 1.

OSIRIS evolved into its own API, creating a layer of abstraction on top of P-BEAST, offering a high-level interface for reading P-BEAST data from archives. While the OSIRIS API is a native C++ interface, Python and Java bindings are offered thanks to the use of SWIG [15], a high-level library wrapping generator, capable of generating the boilerplate code necessary to create the required interfaces. An example can be found in Figure 2.

The third layer of abstraction is the OSIRIS program itself, which is a command line utility that allows to generically read P-BEAST values using the mentioned approach, and offers any user an alternative method for reading P-BEAST data over the existing tools. The command line version of OSIRIS is a simple program using the OSIRIS API itself, only taking care of parsing the command line arguments supplied to the invocation of the program.

While the original goal of OSIRIS was to just read P-BEAST data for the *SAReplay* mechanism, OSIRIS proved to be a valuable tool to other developers interested in reading from the P-BEAST archives for their own applications and for projects unrelated to *SAReplay*. An example is a monitoring application for the HLT farm, currently under development.

3.4. Reading ERS data

In the case of ERS events, the data is archived in an Oracle SQL database, and while the values are read using the standard JDBC driver, no existing interfaces were available to ease the reading procedure. Consequently, an internal API to read ERS events from the data store had to be created, with the option to request a time interval, an ATLAS Run Number, or both. Finally, by using this abstraction layer on top of the JDBC code, each ERS message is read one at a time from the database and a new ERS event is constructed.

3.5. Reading the OKS configuration

Access to the archived system configurations is already provided by the OKS system. The Replay mechanism needs to determine the required version of the OKS configuration to be read from the storage system, download the data, and initialize the subsystem from the backups instead of the ATLAS detector infrastructure. Contrary to IS and ERS events, OKS configuration is not a timeline of events, but a global static set of values. The determination of which version of configuration to use is inferred from the selected time interval.

3.6. Injection of events to the SA

Special consideration had to be taken into account regarding the reconstruction of IS objects from P-BEAST data. When an IS object is updated it always produces a notification to the listeners but, for performance and disk storage reasons, P-BEAST discards repeated updates of the same object with the same value. The only additional piece of information P-BEAST stores attached to each value are two timestamp values: the creation date and the last update date. Reading data from P-BEAST must take this detail into account. The approach taken to mitigate this issue was to generate a variable and configurable repeated set of events, starting from the initial timestamp until the timestamp of the last update.

Once the data has been read, the events from both ERS and IS are sorted in a single timeline, and one at a time a new Esper event is created. The strategy to mix the heterogeneous set of values in a single timeline follows the same algorithm as detailed in Figure 1. A min-binary-heap of sequences is kept, having each sequence internally sorted.

EPL directives allow the definition of statements matching patterns of events within timing constraints, for example to count the number of events in the last 20 minutes. Since in the Replay mechanism events are being injecting as fast as possible for performance reasons, it is also necessary to fake the passage of time to allow the execution of statements in an environment which resembles the real execution environment. There are time intervals with associated data that, for example, spans over 20 hours of ATLAS operations, and it would be impractical to wait 20 hours for the testing of directives, especially if the user is only interested in a subset of the data. The Esper engine allows the definition of a custom source of timing, so that time can advance at the speed and granularity that the user needs. This feature is used to fake the passage of time while injecting events from the past, allowing to test time dependent directives in variable periods of time.

4. Web Application

After feedback from the users, it was realized that further simplification of the process could be made. Users had to go through several steps to be able to execute the testing mechanism, and for this reason a web application for testing the SA directives was conceived as a mean to ease the access to the tool.

The web application consists of a simple user interface where the user should enter an identification of the execution, the time interval to read data from, the subset of P-BEAST data as a list of object data types, and the user defined directives to be tested. The user can also define a time limit for the execution, having by default the value of 1 hour of wall-clock time. If the execution takes more time, the running process is killed.

Written entirely in Python using the Django Framework, the application has the modules to interact with the user over an HTTP connection with the database where the execution information is stored, and with a background worker task which takes care of the actual execution of the process.

When users fill the interface fields to begin a new execution, the data is stored in the database. Then, one of the worker process running on the server in the background selects the next unprocessed job from the database, creates a new runtime environment for running the *SAReplay* mechanism, and begins the process while monitoring its output and status. The background process may terminate the execution under several conditions; the usual one being when the execution has finished. The worker can also decide to terminate the process because the timeout has been reached, or too much disk space is being used by the execution by producing a large number of alerts.

Each execution is uniquely identified by an identifier; for every execution all relevant data and files are stored in the database permanently, and users are allowed to peek and seek previous executions, not only of their own ownership but also from other users.

5. Conclusion

In this paper we have presented a complete testing framework for the ATLAS Shifter Assistant that allows experts from different domains to insert their knowledge into the application and verify the correctness of the statements. The reproducibility of the tests allows collaboration among colleagues while developing code and also permits to introduce a formal (and clear) approval procedure for code that is ready to be put into production. Enabling the distribution of EPL statements development to a varied community of experts will hopefully allow us to overcome the typical problem of knowledge based management: keeping it up-to-date and alive.

The availability of good tools to help in the development process is essential in the development effort. Having a context for executing directives which allows to iterate over different variations of the same directives and/or data sources is an important asset in the development experience. Keeping a test environment in a state capable of unambiguously reproducing interesting situations allows to fine-tune the conditions for the generation of alerts, and helps to keep a documentation with past variations of directives.

Despite the fact that several implementation details are specific to this particular application, the approach of providing extensive test coverage by feeding input stemming from the real system can be generically applied to many data driven systems. Keeping a history of events in external storage for a later reconstruction, mocking the data input and alerts output, applying time re-ordering and time progressing, and providing utility software to facilitate the procedure of restarting the engine from fake or previous events is a scheme which may be adopted in other systems with similar problematics.

The success of a testing tool heavily relies on its ease of use: ideally, the tester should not need to know any detail about the internals of the testing mechanism. For this work a web based user interface was chosen: the tester only needs to provide the code to test and a time interval containing the archived data that are relevant for the testing session. While the test input is now self explanatory, additional effort will be put in the future to improve the representation of the test output, and help the tester to understand any issue that occurred during testing. While the task of software testing can be considered intensive and time consuming, the web application for the *SAReplay* project allows the unattended execution of tasks, keeping a queue of awaiting tasks and a complete history of past executions, allowing the user to run the testing procedure at any time. It was also found that the web application allows easier collaboration among developers, and promotes the communication between experienced developers already familiar with the EPL paradigm and newcomers who want to write their own directives.

References

[1] ATLAS Collaboration 2008 *J. Instrum* **3** S08003

[2] ATLAS Collaboration 2003 *ATLAS Technical Design Reports*

[3] Kazarov A, Lehmann Miotto G and Magnoni L 2011 The AAL project: Automated monitoring and intelligent analysis for the ATLAS data taking infrastructure Tech. Rep. ATL-DAQ-PROC-2011-053 CERN Geneva URL `http://cds.cern.ch/record/1403070`

[4] EsperTech 2014 Esper complex events processing URL `http://www.espertech.com/esper/index.php`

[5] Soloviev I and Sicoe A 2014 New persistent back-end for the ATLAS online information service Tech. Rep. ATL-DAQ-PROC-2014-008 CERN Geneva URL `http://cds.cern.ch/record/1703443`

[6] Python Software Foundation 2014 Python URL `http://www.python.org`

[7] Django Software Foundation 2014 Django URL `http://www.djangoproject.com/`

[8] Kolos S, Boutsioukis G and Hauser R High-performance scalable information service for the ATLAS experiment Tech. rep.

[9] dos Anjos A, Beck H, Kolos S, Gorini B and Wiesmann M Error handling and error reporting in TDAQ applications Tech. rep. URL `https://edms.cern.ch/file/459790/1.0/ErrorHandling.pdf`

[10] Jones R, Mapelli L, Ryabov Y and Soloviev I 1998 *Nuclear Science, IEEE Transactions on* **45** 1958–1964

[11] Peters A J and Janyst L 2011 *Journal of Physics: Conference Series* vol 331 (IOP Publishing) p 052015

[12] Object Management Group 2014 Corba URL `http://www.corba.org/`

[13] Gamma E, Helm R, Johnson R and Vlissides J 1994 *Design patterns: elements of reusable object-oriented software* (Pearson Education)

[14] Stroustrup B 2013 *The C++ programming language* (Pearson Education)

[15] SWIG 2014 Swig URL `http://swig.org/`

[16] Anders G, Avolio G, Lehmann Miotto G and Magnoni L 2014 Intelligent operations of the data acquisition system of the ATLAS experiment at the LHC Tech. rep. ATL-COM-DAQ-2014-103

[17] Runeson P 2006 *Software, IEEE* **23** 22–29

[18] Lee S and O'Keefe R M 1994 *Systems, Man and Cybernetics, IEEE Transactions on* **24** 643–655

[19] O'Keefe R M and O'Leary D E 1993 *Artificial Intelligence Review* **7** 3–42