ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA178

A FRAMEWORK FOR ONLINE ANALYSIS BASED ON TENSORICS EXPRESSIONS AND STREAMING POOL

A. Calia, M. Hruska, M. Gabriel, K. Fuchsberger, M. Hostettler, M.P. Pocwierz,
CERN, Geneva, Switzerland

Abstract

Among other functionalities, the Tensorics library provides a framework to declaratively describe expressions of arbitrary values and resolve these expressions in different contexts. The Streamingpool framework provides a comfortable way to transform arbitrary signals from devices into long-living reactive streams. The combination of these two concepts provides a powerful tool to describe modules for online analysis. In this paper we describe this approach, elaborate on the general concepts and give an overview of actual and potential use cases as well as ideas and plans for future evolution.

MOTIVATION

Devices of accelerators deliver their measurement data through asynchronous channels to which higher level applications can subscribe. This data can be seen as streams of data items. Recent technology evolution provides concepts on top of which a framework for management of such streams - Streamingpool - was created and is used in operational applications, as described in [1]. Most of the time, subscribing to a single stream is not sufficient, but multiple streams have to be combined and some decision based on a certain logic has to be taken. To base such logic completely on asynchronous operators, as provided by RxJava (the technology used and provided by Streamingpool), turns out to be overly complicated and difficult to read and debug. Additionally, experience from previous application developments at CERN (e.g. the Software Interlock System [2]) showed that it was favorable to base such analysis on immutable snapshots.

When work was started by the BE-OP-LHC software team on a new system for LHC (Large Hadron Collider) injection diagnostics, exactly these challenges arose. The goal was to be able to formulate conditions in a way which could be read and understood by non-programmers and at the same time provide the necessary comfort (e.g. IDE support and code completion) for people who have to formulate such conditions. Further, as good results were already achieved with a similar approach for powering test analysis in the LHC [3,4], it was decided to aim for a Java internal Domain Specific Language (DSL).

This finally led to the analysis framework described in this contribution. It is built in a modular way and consists of the following components, which can be used alone or together, depending on the usecase:

- **Streamingpool** [1] provides an abstraction to streams of data coming from accelerator devices and, in the context of the analysis framework, is used for all kind of asynchronous processing which is required to build the snapshot which then is passed on to the real analysis logic (e.g. triggering, buffering and mapping of streams).
- The **Analysis DSL** is based on (and an extension to) the DSL provided by the Tensorics library [3]. It describes the logic to apply to the data and can e.g.be used standalone, for example to analyze static data (e.g. by pulling from the LHC logging service).

In the following sections these components are described in further detail.

STREAMING POOL

Streamingpool is an open-source framework [1,5] that abstracts the way long-living reactive streams are discovered, created and managed.

In the Streamingpool, each stream is uniquely identified by a `StreamId`. A `StreamId` provides an abstraction over what the developer wants to get as stream (e.g. a stream of data from an hardware device). Given a `StreamId`, one can discover the associated stream (returned as a `Publisher<T>`) using a `DiscoveryService`. This service queries the Streamingpool for the stream that is identified by the provided `StreamId`. If the stream was already discovered before the Streamingpool then returns the same `Publisher<T>`, otherwise triggers its lazy creation and then caches it for subsequent requests.

ANALYSIS DSL

The logic for an online analysis is described by extending the `AnalysisModule`. The `AnalysisModule` provides a custom Java-based DSL for expressing the analysis logic. This feature gives the possibility to the developer to implement complex logic while having the full flexibility of the Java programming language. It also makes the analysis type-checked as it is possible to check for errors at compilation time and it gives full auto-completion capabilities during the development.

Assertions

The basic building blocks for an analysis are assertions. They are used for specifying conditions to check during the analysis. As an example, consider a user wants to assert that "protons are **produced**", but this condition should only be taken into account when "protons are **requested**" (because only then the condition makes sense).

This could be formulated through the analysis DSL as shown in Listing 1.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA178

Listing 1: Assertion inside an Analysis Module code example.

```
whenTrue(PROTONS_ARE_REQUESTED)
  .thenAssertBoolean(PROTONS_ARE_PRODUCED)
    .isTrue();
```

Here, both constants (`PROTONS_ARE_REQUESTED` and `PROTONS_ARE_PRODUCED`) refer to `StreamIdBasedExpressions`, as will be explained in a following section.

The example shows that an assertion is in general composed of two boolean expressions: the condition and the precondition. The precondition is formulated in the `whenTrue(...)` clause ("protons are requested" in the above example). The condition representing the real business logic is then specified using the `thenAssert...(...)` family of methods. If preconditions are omitted then they will be considered as being always `true`.

Based on this defined logic, each assertion can then result in one of the following states, whenever the analysis is evaluated:

- `SUCCESSFUL`: if the boolean condition yielded `true` as result.
- `FAILURE`: when the condition has `false` as result.
- `NON_APPLICABLE`: if the precondition expression was evaluated to `false` (masked).
- `ERROR`: when an expected error occurred during the evaluation of the condition or precondition expression.

Analysis Result

After evaluation, each analysis module also produces an overall result, which represents the summary of all assertion results contained in the module. Possible result values are:

- `SUCCESSFUL`: all the assertions in the module are `SUCCESSFUL` or `NON_APPLICABLE` (masked).
- `FAILURE`: at least one assertion yielded value `ERROR` or `FAILURE`, so the overall analysis is considered a failure.
- `ERROR`: an unexpected error happened during the evaluation of the `AnalysisExpression`.

Evaluation Strategies

Typically, an online analysis has streams as inputs (identified by `StreamIds`, as described before). A specialized analysis module, called `StreamBasedAnalysisModule`, is provided for further extension to cover these cases. This class provides additional functionality to the DSL for working with streams of data. The exact behaviour of an online analysis module depends on the sub-type of `StreamBasedAnalysisModule` from which it inherits. The following three behaviour-types are currently available:

- The logic of a `ContinuousAnalysisModule` is evaluated each time any input stream of the analysis receives new data, the analysis is recalculated with the latest value of all the other streams. Further parametrization of the behaviour is neither possible nor required.
- Logic from a `TriggeredAnalysisModule` is evaluated each time a particular stream emits an item. The author of the module can specify this stream as part of the DSL of this type of analysis module. As data of the input streams arrive, only the latest value per stream is kept. When the trigger stream yields a value, the analysis is then evaluated using the latest values for the input streams that were previously saved.

Listing 2: Syntax for parametrizing a triggered analysis.

```
new TriggeredAnalysisModule() {
  {
    triggered().by(aTriggerStreamId);
  }
});
```

- Finally, the `BufferedAnalysisModule` specifies analyses that act on buffered input streams specified according to a `BufferEvaluation` strategy. In this case, the DSL offers specific methods for specifying when the buffering should start and end. A key class for buffered online analyses is the `BufferedStreamExpression`, which is a node in the analysis graph that takes a `StreamId` as parameter. During the evaluation of the analysis module, a `BufferedStreamExpression` yields the buffer containing the values that have be saved from the stream of data identified by the specified `StreamId`. In this way, the buffering and the stream definition (`StreamId`) are decoupled. An example DSL snippet for specifying buffer start- and end-streams looks like this:

Listing 3: Example of a BufferedAnalysisModule.

```
new BufferedAnalysisModule() {{
 buffered()
  .startedBy(startStreamId)
  .endedOnEvery(endStreamId)
  .or()
  .endedAfter(Duration.ofSeconds(10));
}});
```

TENSORICS EXPRESSIONS

The expression part of the analysis framework is based on Tensoric Expressions, which are provided by the Tensorics open-source project [6]. This part of the library provides a Java DSL to describe operations on input data which can later be applied to incoming data in different contexts. Technically speaking, a Tensorics Expression is a node in a direct acyclic graph (typically a tree in simple cases, as e.g. illustrated in Fig. 1) which can have children and can be resolved to a specific value. The main advantages of this approach, compared to directly executing Java code are the following:

- The graph is an immutable Java object and can be serialized (with some restrictions). This makes it possible that e.g. the analysis rules can be sent at runtime to another Java process (potentially on another host). This enables distributed computing without redeployments.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA178

- The information contained in the graph can be accessed easily for further processing. This feature is used within the analysis framework e.g. to automatically construct meaningful names for the assertions. Further it will make it possible to create generic GUI components, which can e.g. display detailed information on the reasons why certain assertions failed. This is a similar approach as taken in [3].
- Since the full information of the graph is available, it is trivial to determine the required input data (the leave nodes), which is very hard if plain Java execution would be used. This is a key-concept for the described analysis framework: A dedicated `StreamFactory` determines the required input streams from the graph and passes them on as input to the resolution mechanism.
- The resolution mechanism itself can be optimized without changing (or even recompiling) the graph logic. This will be described in the following section.

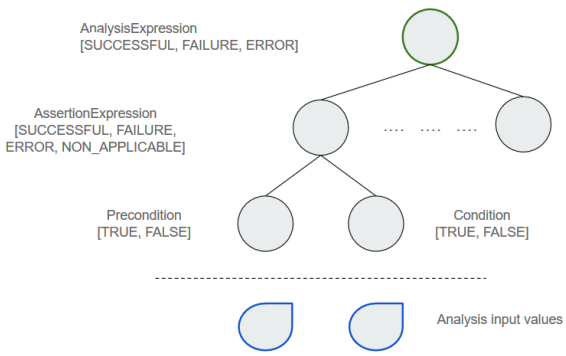

Figure 1: Tree structure for an online analysis.

An Example

First of all it shall be noted that, although the expression mechanism comes from the Tensorics library, we are not using any tensors here. Actually, the expression mechanism is generic, so that it can in principle be extended to any operation required.

Consider calculating the average bunch intensity from the total beam intensity and the number of bunches. The corresponding logic would look somehow like this:

Listing 4: Example of a TensoricsExpression.

```
Expression<Double> totalIntensity;
Expression<Double> numberOfBunches;
/* Creation omitted */

Expression<Double> avgBunchIntensity =
  TensoricDoubleExpressions
    .calculate(totalIntensity)
    .dividedBy(numberOfBunches);
```

After executing this piece of code, the variable `avgBunchIntensity`, would represent a tree with one operation (division) and two operands as leaves (total intensity and number of bunches).

The resolution of the actual value of such an expression is delegated to a `ResolvingEngine`. This would in the simplest case look like this:

Listing 5: Resolving an Expression.

```
ResolvingEngine engine;
/* Creation omitted */

Double result = engine
  .resolve(avgBunchIntensity);
```

The resolving engine is an object that is able to walk through the graph and resolve the nodes. It does so by the use of so-called resolvers, each of them capable of resolving a specific type of expression to its value. The detailed algorithm is exactly the same as described in [4] and is thus not repeated here. However, compared to DSL used in the accelerator test tracking framework (AccTesting) [3, 4], the tensorics expression language has the following improvements:

- The resolved type can be any Java type, while in the AccTesting language required special wrapper types.
- The tensorics language provides support for several steps our of the box (e.g. calculations based on scalars, tensors and iterables), which are very hard to implement in the AccTesting DSL, because of the limiting data types used in there.

Any node of the graph can carry a value (called `ResolvedExpression`) or be unresolved (`AbstractDeferredExpression`). In the latter case, the value of the node is calculated by one Tensorics `Resolver` when the graph is resolved. The concept `Resolvers` provides the necessary decoupling of expression and corresponding code execution: For example, if a node is implementing the sum operator, a custom `Resolver` has to query the result of the operands and then return the sum of the values. In the above example, the `totalIntensity` variable could e.g. be assigned to a particular expression that represents a variable in the logging system and a point in time. A corresponding `resolver` would then pull the corresponding value from the logging system when queried by the resolving engine.

Resolving Context

During the resolution of the final graph result, the nodes are resolved bottom up (leafs first) and all sub results are collected in an object called `ResolvingContext`, so that they are available for the resolvers of the parent node. The resolving engine provide an additional override to the `resolve(..)` method, which allows to feed in an initial context. The context can be seen as a simple map from expression to the resolved value of it and also can be constructed as such. If all the inputs are provided, then the resolving engine can do the calculation with its internal resolvers only (without any special one - e.g. to query the logging database). In this case the above example reduces to a simple calculation:

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA178

Listing 6: Example of a TensoricsExpression.

```
Expression<Double> t =
            Placeholder.ofName("t");
Expression<Double> n =
            Placeholder.ofName("n");;

Expression<Double> avg =
 TensoricDoubleExpressions
            .calculate(t).dividedBy(n);

ResolvingContext ctx =
 ResolvingContext.of(t, 2.2e11, n, 2.0);

ResolvingEngine engine =
 ResolvingEngines.defaultEngine();

Double result = engine.resolve(avg, ctx);
/* results in 1.1e11 */
```

EXTENSIONS TO STREAMING POOL

The Analysis framework uses exactly the feature of a prefilled context - as described in the previous section - in order to bridge Streamingpool and Tensorics Expressions. The required extensions to streamingpool are encapsulated in the streamingpool-ext-analysis project [7]. The main parts of this extension are the following:

- A `StreamIdBasedExpression<T>` can be used as a node in the DSL and carries as payload a corresponding `StreamId<T>`.
- The `AnalysisStreamId` has as payload an expression tree which describes the analysis logic. Its leaves potentially can be instances of `StreamIdBasedExpression<T>`
- Several different implementations of `StreamFactory` are responsible for determining the required input streams from the expression tree of the `AnalysisStreamId` , looking up the corresponding streams, applying the required buffering strategy, correctly prefilling the `ResolvingContext` and finally querying the `ResolvingEngine` for the final analysis result.

In other words, the pre-filled `ResolvingContext` represents a snapshot of the input data to be analyzed. This approach makes the real logic synchronous and reproducible. An extension, envisaged for the future, is also the possibility to record the input data and either replay later the snapshot only or even the individual streams. This would make it possibly to debug logical issues or even inconsistencies in the asynchronous behaviour.

TECHNOLOGIES

The analysis framework is based on the latest technologies available on the Java world. In addition to the aforementioned Tensorics Expressions and Streamingpool, Spring [8] is used for dependency injection and reactive streams [9] are used for handling inputs (using RxJava [10] as Java implementation).

The code of the online analysis framework makes heavy use of the latest Java features: Default methods in Java interfaces are used for composing utility classes for making unit testing as easy as possible. Furthermore, the public API of the framework has been crafted to be compatible with Java 8 lambdas where possible, in order to optimize the coding experience of the user.

SPRING BOOT STARTER PROJECTS

The online framework presented in this paper makes use of Spring for dependency injection. In order to ease the adoption of the online analysis framework, a set of Spring Boot project have been created ([5]). Given the way Spring Boot works, just by adding the starter project as a dependency in a Spring Boot application, the developer will have a pre-configured online analysis environment ready to be used. Customizations are still possible by overriding the default values were required by any particular scenarios.

APPLICATIONS

The presented online analysis framework was mainly driven by the development of the LHC Filling Diagnostic [11]. This is a software that evaluates the status of the CERN's accelerators control system in order to provide a diagnostic for the LHC injections. In order to achieve this goal, it subscribes to many signals from the control systems of LHC and SPS accelerators and compares the signals against specific rules. The result of the online analysis is the status of a LHC injection and an explanation in case it fails. The tool is currently in beta version. Despite this, it is currently being of help to the LHC operations crew and for statistics gathering for the LHC injection performance.

Further applications are under investigation, e.g. for a continuous check of a new variant of the LHC Beam Dump Loop or several new displays in the LHC to assist the operations crew in quick decisions. Another idea is to use this as a base for future developments towards a more state-machine driven operation of the LHC.

SUMMARY AND OUTLOOK

In this paper a new framework for online analysis was presented. Based on Streamingpool and Tensorics frameworks, it provides an easy way of specifying and managing analysis of signals. At CERN these kind of analysis is especially useful to monitor the state of the accelerators control systems as well as the state of the hardware devices.

The framework features a powerful and extensible Java-based DSL that a developer can use to create custom analyses. With the concept of assertions and evaluation strategies, the analysis can be adapted to a large variety of use cases.

There are numerous ideas for improving the framework. The most important one of them is the notion of time in the analysis buffers. At the moment, when a developer selects a buffered evaluation strategy, the buffer is provided as a list of items. In this situation is not easy to correlate data between signals that arrive at different moments in time. A

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA178

prototype version of this improvement is under development and the details will be presented on a different paper.

ACKNOWLEDGMENT

The development of the online analysis framework is the result of a cross-section collaboration at CERN. A special thank is given to section BE-LHC-OP, TE-MPE-MS and BE-CO-APS for the comments, reviews and contributions.

REFERENCES

[1] A. Calia, K. Fuchsberger, M. Gabriel, M.-A. Galilée, G.-H. Hemelsoet, M. Hostettler, M. Hruska, D. Jacquet, J. Makai, "Streaming Pool - Managing Long-Living Reactive Streams for Java", ICALEPCS2017, Barcelona, Spain (2017), THPHA176

[2] L. Ponce, J. Wenninger, J. Wozniak, "Operational Experience with the LHC Software Interlock System", ICALEPCS2013, San Francisco, CA, USA (2013), MOPPC069

[3] M. Audrain et al., "Using a Java Embedded DSL for LHC Test Analysis", ICALEPCS2013, San Francisco, CA, USA (2013),THPPC079

[4] K. Fuchsberger et al., "Concept and Prototype of a Distributed Analysis Framework for LHC Machine Data", ICALEPCS2013, San Francisco, CA, USA (2013), TUPPC026

[5] streamingpool, https://github.com/streamingpool

[6] tensorics-core, https://github.com/tensorics/tensorics-core

[7] streamingpool-ext-analysis, https://github.com/streamingpool/streamingpool-ext-analysis

[8] Spring IO, https://spring.io

[9] Reactive Streams, http://www.reactive-streams.org

[10] RxJava, https://github.com/ReactiveX/RxJava

[11] A. Calia, K. Fuchsberger, G.H. Hemelsoet, D. Jacquet, "Development of a New System for Detailed LHC Filling Diagnostics and Statistics", IPAC2017, Copenhagen, Denmark (2017),TUPIK088