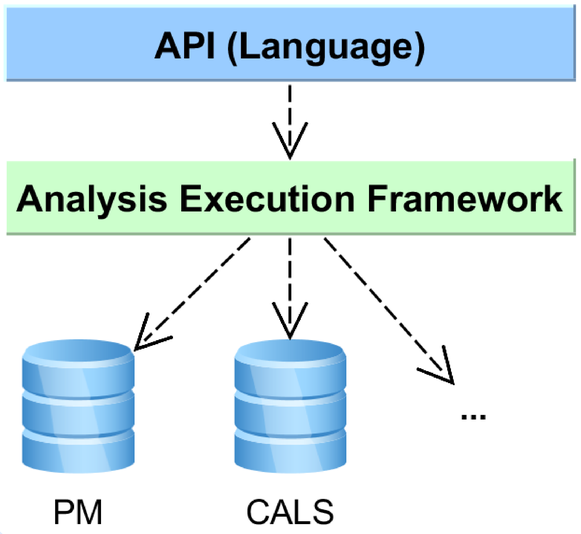
Concept and Prototype for a Distributed Analysis Framework for the LHC Machine Data

K. Fuchsberger, J.C. Garnier, A.A. Gorzawski, E. Motesnitsalis (CERN, Geneva, Switzerland)

Overview



Motivation

The LHC produces about 50 TB of diagnostics data per year, mainly stored in two systems:

- **LHC Post Mortem System (PM):** Stores data at a high resolution over short time ranges.
- **Common Accelerator Logging Service (CALS):** Provides continuous logging of equipment signals all around the LHC.

Problems:

- Hard to correlate data
- Slow Data extraction
- High code duplication

References

[1] K. Fuchsberger et al., "Automated Execution and Tracking of the LHC Commissioning Tests", proc. of IPAC12, New Orleans, LA, USA.

[2] D. Anderson et al., "The AccTesting Framework: An Extensible Framework for Accelerator Commissioning and Systematic Testing", these proceedings

[3] D. Andersen et al., "Using a Java Embedded DSL for LHC Test Analysis", these proceedings.

[4] http://akka.io

[5] E. Motesnitsalis, "Using Akka Platform in Unidentified Falling Object Detection on the LHC", CERN TE Notes, CERN, Geneva, 2013.

Abstract

The Large Hadron Collider (LHC) at CERN produces more than 50 TB of diagnostic data every year, shared between normal running periods as well as commissioning periods. The data is collected in different systems, such as the LHC Post Mortem System (PM), the LHC Logging Database and different file catalogs. To analyze and correlate data from these systems it is necessary to extract data to a local workspace and to use scripts to obtain and correlate the required information. Since the amount of data can be huge (depending on the task to be achieved) this approach can be very inefficient. To cope with this problem, a new project was launched to bring the analysis closer to the data itself. This paper describes the concepts and the implementation of the first prototype of an extensible framework, which will allow integrating all the existing data sources as well as future extensions, like hadoop clusters or other parallelization frameworks.

Requirements

Basic Requirements:

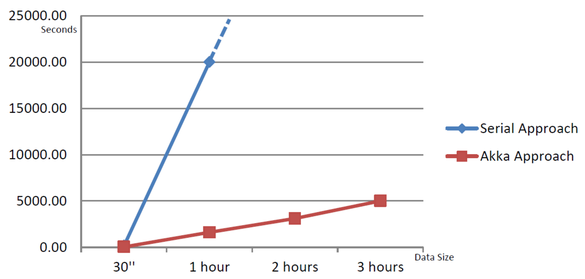
- Calculations close to the data
- Horizontal Scalability

Should also handle:

- Data incompleteness
- Data invalidity
- Mathematical operations
- Physical units
- Error propagation

Parallelization

Prototype using Akka [4]:



- Gained factor of 20 (parallel data extraction) on one node [5].
- Next steps:
  - Parallization of Dispatcher
  - Prototype on real cluster

eDSL and Execution

Very flexible layer of execution:

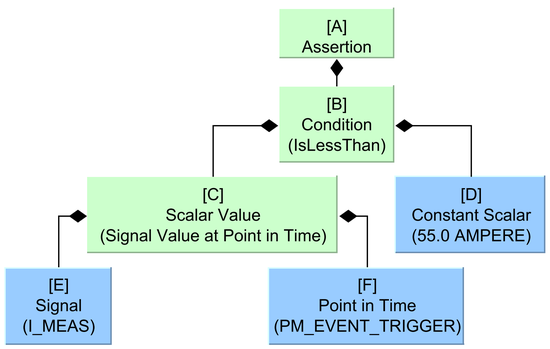
- Start with a very simple solution
- Later optimize execution without any impact on the language level

An Example:

Listing 1: Simple Assertion

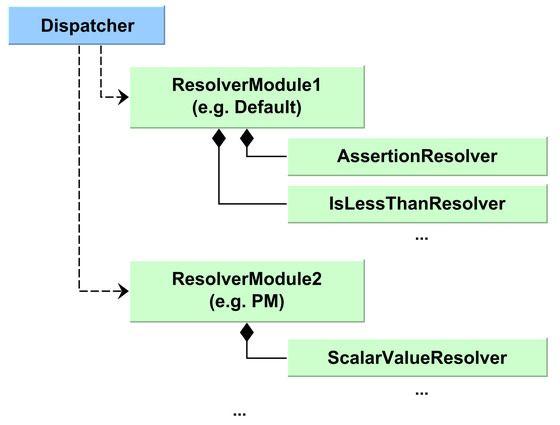
```
assertThat(I_MEAS).isLessThan(55.0, AMPERE).
    at(PM_EVENT_TRIGGER);
```

Produced Expression Tree [3]:



- Square brackets ([]): labels for the nodes
- Blue: resolved (known values)
- Green: unresolved (to be calculated)

Tree Resolving



Algorithm:

1. Starting from root node [A], dispatcher queries each resolver: `canResolve(..)`. A resolver can resolve a given node,
   - if signature of the `invoke(..)` method matches the node
   - and if `canInvoke(..)` method returns `true` for the node.
2. Dispatcher remembers potential candidate resolvers in a list.
3. If no resolver found: recursive invocations (node [B] in example).
4. Else: Dispatcher stops the iteration through the tree in this branch (node [C] in example).
5. Dispatcher selects one of the candidate resolvers per node, invokes it and rebuilds the tree.
6. After all resolvers returned: loop starting at item (1), until all nodes resolved.

- Easy to add optimized resolvers.
- Potential Improvements:
  - Parallelization of resolvers (5).
  - Machine learning for resolver selection.
  - Caching of intermediate results.