USING A JAVA EMBEDDED DOMAIN-SPECIFIC LANGUAGE FOR LHC TEST ANALYSIS

M. Audrain, K. Fuchsberger, J.C. Garnier, R. Gorbonosov, A.A. Gorzawski, A. Jalal, J. Suchowski, P.C. Turcu, M. Zerlauth, CERN, Geneva, Switzerland

Abstract

The Large Hadron Collider (LHC) at CERN requires thousands of systems to work in close cooperation. All these systems have to be tested during Commissioning and after interventions on them. Starting from the experience of Hardware Commissioning, the execution of such tests were already automated to a high degree. The remaining time in commissioning campaigns is now spent in analyzing test results, which is done manually to a certain extent. To improve this situation, a new project was launched which aims to automate the analysis of such tests as much as possible. For this purpose, a dedicated Java embedded Domain Specific Language (eDSL) was created which allows system experts to describe analysis steps in a simple way. The execution of these checks finally can produce, along with the simple decisions on the success of the tests, plots for the experts to quickly track down the source of problems exposed by the tests. This paper explains the concepts used and the future vision of this first version of the eDSL.

INTRODUCTION

During a LHC hardware commissioning campaign, about 10000 tests are run to determine if the machine is safe for operation. The system which orchestrates the test runs is the Accelerator Testing (AccTesting) framework [1]. A test within AccTesting consists of the following three steps:

- The *test execution* performs a sequence of tasks on a system under test, for example applying a specific Current to a magnetic circuit.
- The *test analysis* validates the results of the test by analyzing the data recorded during its execution.
- The *test signing* allows experts to stamp the two previous test steps with one or several signatures, ensuring that the hardware device is behaving properly with regard to the performed test.

With the experience gained over several years and campaigns of hardware commissioning, it was found that the most time spent in those campaigns concerns the second test step: the analysis of the data read out from tests execution. This analysis can be time consuming because it is usually performed manually by experts or using semi-automated LabView analysis programs. Moreover, those LabView programs are often written and maintained by one expert, with no version control over them. To automatize the analysis execution and ensure analysis scripts maintainability, the analysis framework project was launched. As an essential part of this framework an eDSL was introduced which is called Analysis language in the following.

The first section of this paper will present the motivation and requirements for an automated analysis framework, which led to the creation of an eDSL. The second section will present the eDSL's basic concepts and their application for our analysis language. The third section will present the processing of the language, from the script writing to its calculated result. Finally, the last section will present the detection of syntax errors provided by the analysis framework.

MOTIVATION

The analysis framework's primary need was to provide a user-friendly Application Programming Interface (API) for the system experts to formalize an analysis script to be performed on the data read out from the execution of a specific test on a specific hardware device. Using this API, the experts would write the different assertions (statement that checks that a predicate is true) to be executed for the analysis to be considered as valid. The analysis scripts should be version controlled so any change is tracked and the versions can be shared between experts.

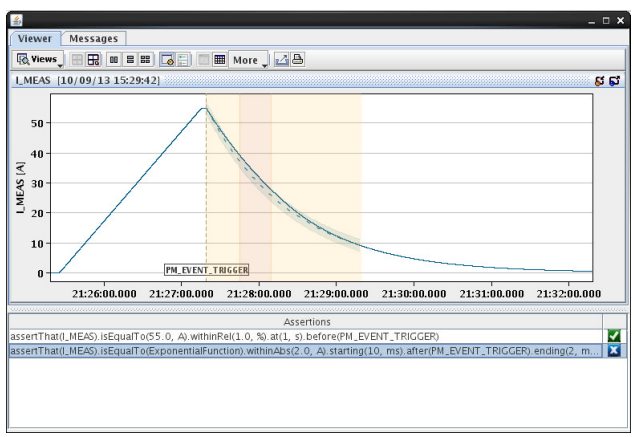

Figure 1: The display of an analysis result consists of a list of assertion lines, and a chart showing the detail of the selected line. The graph highlights the area covered by the assertion in orange and the failing part in red.

All in all, this API's main goal is to take care of mathematical comparison and calculation so the hardware expert can focus on the formalization of the analysis. Additionally, hardware experts are not necessarily software developers, so the analysis framework API should be self-

ISBN 978-3-95450-139-7

explanatory and readable enough to ensure the maintainability of the scripts over time.

Although the language should be modular enough to allow calculations in future versions, its primary requirement in the context of AccTesting is the verification of test results. Therefore, the following requirements were identified:

- The ability to define conditional checks called assertions. These assertions have to ensure that a function of time is comparable to a value at a single moment or to another function of time or over a time interval.
- The use of a graphical user interface (GUI) component to display the signals used in the assertions and the results of those assertions.

```
public class MyModule extends AnalysisModule {
    {
        assertThat(I_MEAS)
            .isEqualTo(55.0, AMPERE)
            .withinRel(1.0, PERCENT)
            .at(1, SECOND)
            .before(PM_EVENT_TRIGGER);

        assertThat(I_MEAS)
            .isEqualTo(EXPONENTIAL_FCT)
            .withinAbs(2.0, AMPERE)
            .starting(10, MILLI(SECOND)).after(PM_EVENT_TRIGGER)
            .ending(2, MINUTE).after(PM_EVENT_TRIGGER);
    }
}
```

Figure 2: Example of script written using the analysis language.

From those requirements, multiple solutions were considered for the analysis language implementation. For example, the definition of a model using XML and its concepts of language transformation could have fit the bill. Finally, most of the researched solutions were pointing to the use of a Domain Specific Language (DSL). Defining a DSL can be tedious because it requires not only to define the language, but also the compilation and interpretation of this language. However, a DSL embedded in an existing language (eDSL) is more reasonable to define as it uses all the power the host language can offer, and therefore it is simply a new API for this base language. Because all of the control system environment at CERN is written in Java, it was an obvious step to use Java as a host language for the eDSL. Further, the Java features (Generics, static imports) and the tools around it (Eclipse IDE) make it easy to create an eDSL without re-implementing basic functionalities.

BASIC CONCEPTS OF THE ANALYSIS LANGUAGE

Analysis domain concepts As shown in the plot part of Fig. 1, typical recorded data of a test consists of a ramp up, a flat top and a fast decay. This data also contains one or more events called Post Mortem triggers that can be seen as analysis reference time markers.

The basic concepts and objects the language has to manipulate can be listed as the following:

- A **Signal** is a discrete (non-continuous) function with the dimension of time as X values and any physical dimension as Y values.
- A **Point in time** is a marker of a given moment in time. An Event can be considered as a point in time within a defined context. In the context of a test execution, we can define the Post Mortem trigger as an event. In some cases, an event can have several occurrences during a specific context.
- A **Time range** can be considered as the period between two inclusive points in time.
- A **Tolerance** can be optionally defined for a conditional check. This tolerance can be either absolute or relative and consists of a range of values.

eDSL concepts The primary aim of the analysis language is to be readable and writable by non-developer users. On the other hand, our choice was to use an embedded DSL, as outlined in the previous section. As a compromise for those orthogonal needs, we decided to use a **Fluent API** [2]. In particular, for the Fluent API of our analysis language, we chose the following concepts:

- **Method chaining** consists of having the current object modifiers returning the host object carrying the next element of the language syntax. Using a set of **Progressive interfaces**, one can enforce the syntax tree so the user can only use the mandatory elements provided all along the chain.The use of this concept is shown in Fig. 2. One drawback of method chaining is that it is against good practices of command-query APIs, defining that queries on an object should only return a value and commands should modify the object state without returning anything. Method chaining is used to ease the readability and the usage of our language.
- **Expression builders** are used to decouple the fluent API building logic from the semantic model execution logic. The principle of the expression builder is equivalent to the builder pattern. In our case, the set of progressive interfaces used to constrain the method chaining are implemented to use several expression builders. The model is then built by chaining the calls of the `build()` methods of the different expression builders.
- The **Semantic model** represents the same subject as the API, meaning that all parts of the language syntax can be transformed into one or multiple model objects. This model can be seen as immutable domain objects populated by the language using the expression builders. In our case, the semantic model is made of immutable objects grouped in a tree structure and the execution of our model consists in walking the tree and resolving each element until the assertion result is found.

ISBN 978-3-95450-139-7

- One more concept that can be useful for the analysis language user experience is **Object scoping**. This concept deals with providing a template for the writing of the user analysis script. With this template, the user has access to the language starting points to write analysis assertions. In the Java language, one can use the **Instance initializer** technique to achieve the template implementation. It consists of using double brackets in the template which behave like the template constructor. This technique reduces the complexity of the embedded language scripts as seen in the example shown in Fig. 2.

Applying the above concepts to our analysis language, the overall syntax can be defined as the following:

```
{INSTRUCTION}.{CONDITION}.{TIME_DEFINITION};
```

The instruction part defines which type of functionality is to be used from the language, either asserting or calculating, and also which signal is to be used as input data for this instruction line.

The condition part of the assertion provides the type of check (comparison to a number, to a function, etc...) that will be performed on the previously defined signal. It allows users to formalize the expected operand (single value, function) and the tolerances to be used for the comparison.

The time definition part defines the time where the previously provided comparison will be applied. If the comparison is successful during this time (either a point in time or a time range), then the assertion statement is considered true.

The dots between the language parts provide a handy delimitation of those parts and represent the nodes of the syntax tree where the user has to make a choice either to use one or another branch of the syntax tree.

Finally, the use of the semicolon formalizes the end of an assertion line where no more syntax branches are available.

LANGUAGE PROCESSING

From an analysis script to an analysis result, the following three processing steps are executed:

- *Java compilation*: An analysis script is created by inheriting the abstract `AnalysisModule` class. The script scope is then limited inside the analysis module. When writing the different assertions formalizing the analysis, the Java compiler will constantly check that the syntax of the language conforms with the defined language API.
- *eDSL compilation*: once the analysis script has been written and compiles from the Java point of view, the framework takes care of calling the `build()` methods of the expression builders. This step, called eDSL compilation in the following, produces the corresponding semantic model. This model is a tree of immutable objects where the main node defines a list of conditional checks to be executed. Each conditional check can be decomposed into three main branches, the actual operand defining the signal the user provided, the expected operand being a continuous function generated from the scalar value the user provided and finally the time during which those operands should be checked. An optional tolerance function can be seen as a fourth branch and is generated from the tolerance the user wants to apply to the assertion. An example of the compilation of the analysis language to a semantic model can be seen on Fig. 3

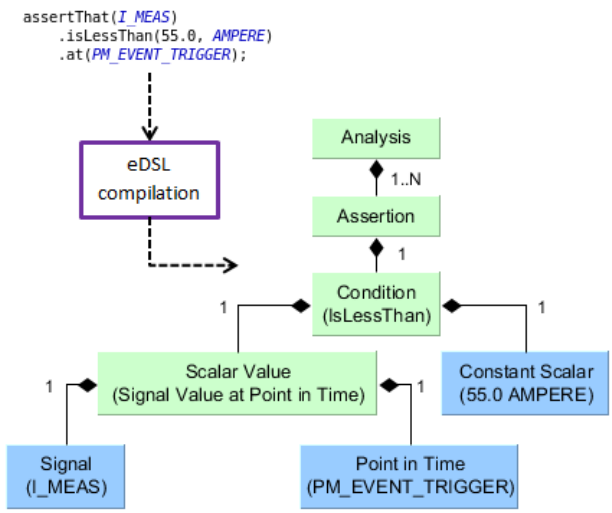

Figure 3: Creation of a semantic model from the compilation of the language.

- *Semantic model execution*: when the semantic model is built, some of its nodes are already resolved, like the expected operand already transformed to a continuous function when its tree node is constructed. Nodes like the actual operand only contain information about the name of the signal and need to be resolved later. These nodes are resolved one by one by a dedicated execution framework, which analyses the expression tree of the semantic model and resolves the tree nodes in the correct order. This framework is currently embedded in the AccTesting server itself and performs all the required data extraction and calculations. There are plans and work ongoing to improve and optimize this framework. Further details can be found in [3].

SYNTAX ERRORS DETECTION

For the assertion language, there are three potential levels of error detection as seen on Fig. 4: script writing time errors (consist of Java compilation errors in the context of our eDSL), eDSL compile time errors (errors occurring when building the semantic model) and run time errors. In the scope of this section, we will focus mainly on the two

ISBN 978-3-95450-139-7

first error detection steps.

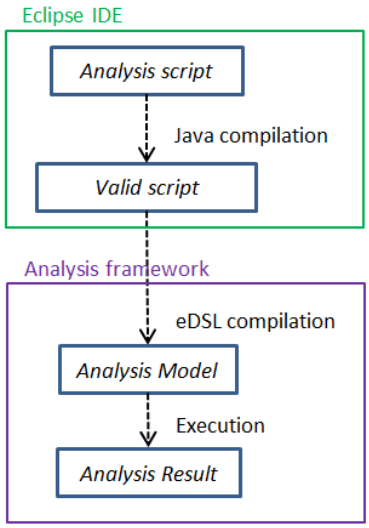

Figure 4: The different language processing steps in the analysis framework.

One of the first design decisions made for our language is that signals or scalar values are strongly coupled with their units. This prevents the user from asserting that a signal of dimension A is comparable to a scalar value with a unit of dimension B and enforces the user to formalize the assertions in an understandable manner.

In order to achieve this design goal, we chose to use the JScience library [4]. Amongst other features, this library provides conversions of units and operations involving different units. Using the `Quantity` interface, one is able to define any type of physical quantity, which then can be expressed in different `Units`. An `Amount` is the combination of a scalar value and its unit. The implementation of the language syntax uses the Java concept of **Generics** to constrain every node of the language syntax with a `Quantity` defined using the JScience library. So when the user starts an assertion with a signal of a certain quantity, the comparison part will restrict the scalar value unit to the same quantity. If the user tries to enforce a different dimension to the scalar unit, an explicit Java compilation error will be raised during the analysis script writing in Eclipse IDE.

Another design decision was to restrict the language syntax as much as possible. It should not be possible to perform multiple calls to a part of the syntax (instruction, condition, time definition) in the same assertion line. Using progressive interfaces introduced previously in the eDSL concepts, we could first restrict the syntax to an understandable chain of methods so every assertion makes sense when read. Only one problem remains: if the user does not terminate an assertion line, the Java compilation step will not raise an error, because even though the syntax is not complete, there is no identifiable java error in the currently defined assertion line. As explained in the previous part, the eDSL compilation will check the completeness of each assertions as well as the validity of the provided parameters (arguments not `null`).

CONCLUSION AND OUTLOOK

The analysis framework has been up and running for several months and scripts formalization is ongoing for some of the automated tests. The analysis framework will be ready to perform the operational analysis for at least one prototype test during the next hardware commissioning campaign in 2014. Further analysis modules will need some more features, which are in the pipeline for 2015. Future plans for the analysis language, which also go beyond simple test verification, concern the implementation of calculation functionalities, which would make this language very useful for more general analysis of operational data of the CERN accelerators. Furthermore, the analysis language could also be used directly in other operational software, where calculations and verifications of stored signals have to be performed. Examples for this are: Post Mortem Modules, Sequencer Tasks or Pre/Post Operational checks of systems.

ACKNOWLEDGMENTS

The authors want to thank the LHC hardware commissioning team for all their feedback. Particular thanks goes to B. Auchmann, S. Rowan, R. Schmidt and Z. Charifoulline for their input on the required features and the language syntax.

REFERENCES

[1] D.Anderson et al., "The AccTesting Framework: An Extensible Framework for Accelerator Commissioning and Systematic Testing", proc. of ICALEPCS 2013, San Francisco, CA, USA.

[2] M. Fowler and R. Parsons, "Domain-Specific Languages", Addison Wesley, 2011.

[3] K. Fuchsberger et al., "Concept and Prototype for a Distributed Analysis Framework for LHC Machine Data", proc. of ICALEPCS 2013, San Francisco, CA, USA.

[4] www.jscience.org

ISBN 978-3-95450-139-7