Message Correlation Analysis Tool for NOνA

To cite this article: Qiming Lu *et al* 2012 *J. Phys.: Conf. Ser.* **396** 012030

Message Correlation Analysis Tool for NOνA

Qiming Lu, Kurt A. Biery, and James B. Kowalkowski
Scientific Computing Division, Fermi National Accelerator Laboratory
P.O.Box 500, Batavia, Illinois 60510, U.S.
E-mail: `qlu@fnal.gov, biery@fnal.gov, jbk@fnal.gov`

Abstract. A complex running system, such as the NOνA online data acquisition, consists of a large number of distributed but closely interacting components. This paper describes a generic real-time correlation analysis and event identification engine, named Message Analyzer. Its purpose is to capture run time abnormalities and recognize system failures based on log messages from participating components. The initial design of analysis engine is driven by the data acquisition (DAQ) of the NOνA experiment. The Message Analyzer performs filtering and pattern recognition on the log messages and reacts to system failures identified by associated triggering rules. The tool helps the system maintain a healthy running state and to minimize data corruption. This paper also describes a domain specific language that allows the recognition patterns and correlation rules to be specified in a clear and flexible way. In addition, the engine provides a plugin mechanism for users to implement specialized patterns or rules in generic languages such as C++.

1. Introduction

The NuMI Off-Axis ν_e Appearance experiment (NOνA) is a long-baseline neutrino oscillation experiment which will study oscillation for the sub-dominant mode of appearance of electron neutrinos in the muon neutrino beam [1]. The data acquisition (DAQ) system of NOνA experiment is complex, consisting of more than 400 distributed but closely-interacting components. 11,520 front-end boards are used to continuously read out the 368,640 detector channels. A timing system with compensation for the large geographic area covered by the Far Detector is used for timekeeping and synchronization of the raw data from different detectors. The raw data first get concatenated, based on the geometrical location of the detectors, in *Data Concentrator Modules* (DCMs). They are then pushed downstream to a buffer farm in which data can be stored for 20 seconds or more while waiting for a trigger or an interesting physics event. The triggered events are then be transferred to permanent storage by a collection of data logger nodes. Finally, there are nodes dedicated for management and coordination of the system, *e.g.*, run control, application manager, message server, and DAQ online monitor [2]. Therefore, to maintain a healthy running state of the DAQ system, it is necessary to have a means of continuously monitoring each participating component and for detecting and reacting, with minimum delay, to abnormalities that might put the system or the quality of data in jeopardy. This system is called the the *Automatic Error Recovery System* (AERS).

The AERS is composed of three functional parts:

- run-state monitoring for each component that generates basic run-state messages of the DAQ system,

- a decision making *rule engine* that analyzes the status logs to extract facts and events from the logs, and
- an error handling supervisor that reacts to the events and carries out proper *actions* based on the type of events it has detected as well as the knowledge it has about this type of error or event.

For the monitoring system, the run status is gathered in two ways. It is initiatively reported by each acting component which does a regular self-check and sometimes initiates queries to direct interacting parts. The run status is also reported by a centralized monitoring system called Ganglia [2, 3], which closely monitors the system through a series of custom-defined matrices and general system conditions such as the CPU and memory usage, *etc.* Either way, the status reports are routed to AERS in the form of a **MessageFacility** log message with predefined message categories and severity levels [4, 5].

For a complex running system with many interacting components, often it is possible for the system to be in trouble, even though each individual component still seems healthy. For example, a detector readout board would not know whether or not it is out of synchronization with the rest of the detector readout boards, unless it has global knowledge of all other readout boards' states—which are not accessible to it. This is where the *correlation message analyzer* comes into play. A correlation message analyzer gains global knowledge by inspecting streams of status and log messages from all components in the system. Correlation analysis of the messages can reveal both individual errors and collective behaviors of the system.

The central part of the correlation message analyzer is a *rule engine*, also called a semantic reasoner, reasoning engine, or simply a reasoner. A rule engine is a piece of software which is able to infer logical consequences (here, the abnormalities in the system) from a set of asserted facts (status from each component) [6, 7]. A *forward-chaining* or *inference* engine is a natural candidate for our purposes since what we need is reasoning from available data. Forward-chaining engines use inference rules to extract additional data [8]. In particular, we are interested in a rule engine that processes so-called reaction or *Event-Condition-Action* (ECA) rules [9].

ECA rules automatically perform actions in response to events, provided that stated conditions hold. They have been used in many settings, including active database [10, 11], personalization and publish/subscribe technology [12, 13], and the specification and implementation of business processes [14]. An ECA rule has the general syntax:

on *event* if *condition* do *actions*

The rule is triggered when an *event* occurs. The rule engine then performs a query against the *condition* to determine whether the *action* needs to be taken.

We begin our paper with a brief discussion on adapting the ECA model to the event/error detection of the NOνA AERS, based on correlation analysis of log messages. We then further extend the model for dealing with status *facts* which are collapsed into a group of components by introducing *extended-facts* and semantics for complex *conditions*. We also describe the domain-specific language we used for specifying an extended-ECA rule on event detection. The NOνA AERS is the driving force of this correlation message analysis tool. But we have made the rule engine part of the product generic—with minimum effort it can be deployed in other experiments or systems which have the need of detecting abnormalities or complex events that would require global knowledge of the system. We also discuss the effort involved in making it generic, such as adding user-defined condition patterns for a rule. Along the way, we discuss directions of further work for the message analysis tool, as the development of the NOνA AERS is still ongoing.

2. Correlation message analyzer

The Message Analyzer operates on a stream of messages delivered by the logging system. An incoming message triggers the evaluative part of the software, which follows a set of predefined rules to determine whether there is an abnormal situation. The NOνA Near Detector generates a large volume of log messages during daily operations. The rate ranges from tens to hundreds of messages per second with bursts of over a thousand. With this large volume of data, care must be given in the design of this component to prevent system overloading and poor response times. To cope with this situation, we do the analysis of messages in two steps.

The evaluation of ECA rules can be cascaded such that, when an ECA rule is triggered for examination of its conditions on an incoming event, the action may in turn trigger new rules for further assessment. This continues until a goal is achieved, or until no more rules are triggered. We designed the message analyzer to have two layers of cascading rules. The first layer, driven by incoming log messages, is responsible for inferring status from messages for each component. The status could be a for a node, for the system, or any statements that can be used for further inferences. We call ECA rules for the first layer as *fact extraction rules* with the inferred status as *facts*. The second layer is built upon basic facts to conclude that a given event or situation has happened or is currently happening in the system. We call the second layer of ECA rules *event extraction rules*, or simply *rules*.

2.1. Log messages

The NOνA online DAQ uses a library called **MessageFacility** for publishing and collecting log messages with various severity levels or topics over a distributed system [2, 4, 5]. **MessageFacility** allows the components of different types or platforms in the DAQ system to report errors or status in a consistent and uniformed manner. A typical log message consists of a message body and metadata that is either specified by the user or generated automatically. The fields that matter to the Message Analyzer are: issuer of the message, severity and category of the message, timestamp of when the message was issued, and, of course, the message body. It is worth noting that the Message Analyzer does not solely rely on the **MessageFacility** for providing the log messages. It is designed so that the current logging system can be replaced, as long it provides similar metadata (examples are syslog and JMS) The interchangeability of the low-level message feed allows the tool to be migrated to other experiments or systems with minimum efforts.

One advantage of using **MessageFacility** as the message feed is its distributed message routing capability. **MessageFacility** uses PrismTech's OpenSplice, an implementation of Data Distribution Service (DDS) for its network transportation layer. DDS provides reliable publisher/subscriber communication model without a centralized server [15]. It has greatly increased the robustness of the messaging system, and makes it possible to have multiple instances of Message Analyzer connected to the network backbone without interference.

2.2. Message Analyzer Facts

Facts are assertions based on observations from log messages. Examples of these assertions are: "sensor 7 reads a high temperature", or "started data taking at 12:07pm". Users provide rules in a configuration file to identify and trap log messages matching certain facts. A fact rule is basically a series of tests which cause tasks such as message filtering and pattern recognition to be carried out against incoming messages. A message is trapped by a fact rule when it passes all tests of the rule, indicating the related "fact" has been recognized or has been observed in the system. To summarize, a fact rule takes the input of log messages, and outputs a flag with values true or false (indicating consistency of fact with log messages).

The Message Analyzer provides the following tests from which users can choose: *1*) *issuer* of the message, *2*) *severity* of the message, *3*) *category* of the message, *4*) *regex matching pattern* for the message body, and *5*) a *frequency test* of the message occurrence within a configurable amount of time. All or any subset of them can be chosen, preferably one that is best suitable for identifying a fact.

During data taking, for example, the NOνA run control system sends out a heartbeat check every other second to every component in the system, and expects responses from all of them. If a component fails to respond, run control sends a log message reporting the situation. If a user wishes to trap the fact "data concentrate module (DCM) had a heart attack," he might employ the following tests on the log messages:

(i) Test if issuer is "RunControl"
(ii) Test if severity level is "Warning" or higher
(iii) Test if category is in "RegularCheck"
(iv) Test if body matches expression ".*DCM.*missed heartbeat"
(v) Test if this message has been seen over 10 times in 60 seconds

If all five tests listed above evaluate true, the system will mark a flag to *true* under the fact "DCM has a heart attack". The flagged outcome of the fact can later be used in error detection rules for helping determine a more complex situation, which will be discussed in the following section.

2.3. Message Analyzer Rules
A single fact can usually provide valuable information in detecting errors or other situations present in the system. There are, however, times that a single fact alone is not enough to identify a problem, or pinpoint its exact location. In some cases, it is also possible that facts are flagged due to false alarms, so multiple sources are needed to confirm the situation. For example, from the single fact "DCM misses heartbeat" it would be difficult to decide whether the problem is due to a dead DCM node, or located at the connection between RunControl and the DCM. It is even possible that the problem is really at the RunControl side, where it fails to receive responses. An inference engine with collective knowledge derived from multiple facts is a more robust and comprehensive way to conclude logical consequences. In our case, this means identifying errors or particular events within a running system. The inference engine can utilize the knowledge of the experimenters, who embody their experience in user-defined error detection rules, which we simply call the *rules*.

Rules are built on facts, which are internal flags of true or false indicating whether the fact has happened or not. So the natural choice is to use a Boolean expression to connect all facts that are relevant to the rule. From the ECA model, this Boolean expression is the *condition* part of a rule. The evaluation of the condition is triggered by state changes (from true to false, or from false to true) in any associated fact; we describe this as the *event* part of the rule. When a condition is evaluated as true, a pre-defined sequence of actions is carried out as the reaction to the observed situation.

When describing a rule, users are required to provide the condition and the corresponding reaction parts, while leaving out the event part. Rules are driven by facts, which are in turn driven by log messages. A simple example rule could be:

if *"DCM heart attack"* and *"during data taking"* then *"send an email to the shifter"*.

Here "DCM heart attach" and "during data taking" are facts obtained from log messages. The expression "send an email to the shifter" is the action to be executed if the condition holds.

2.4. Facts and rules in configuration language
The facts and rules form the knowledge base of the Message Analyzer. They are usually provided in the form of a configuration file, written by people who have detailed operational knowledge of the system. Facts and rules are likely to be updated frequently, especially when

during the development phase of a system. Therefore, a human-friendly configuration language, with easy-to-understand semantics can help alleviate the burden of creating and maintaining a knowledge base for the analyzer. The configuration language we use is called Fermi Hierarchical Configuration Language (or FHiCL) [4, 16]. It is loosely based on the syntax of JavaScript Object Notation (JSON) and allows the creation of nested *name-value* pair lists (or parameter sets) to be used for program configuration. FHiCL provides extended features such as hierarchical structures with heterogeneous data types, parameter substitution, and inclusion of predefined parameter sets. The syntax is simple and human-readable. Within this paper, we refer to blocks of name-value pairs as *datasets*. Each dataset is permitted to have a name.

Within a Message Analyzer configuration file, all facts are enclosed in a dataset called "facts":

```
facts : {
    fact_1 : { ... }
    fact_2 : { ... }
}
```

Every fact inside the facts block must be named uniquely to avoid collision. A fact dataset may include up to five named fields corresponding to five tests (as described in section 2.2), along with an optional description field. The fields are be described as follows.

(i) *source*: contains an array of strings listing the issuer of the message
(ii) *severity*: a string containing one of "DEBUG", "INFO", "WARNING", or "ERROR", that specifies the severity threshold
(iii) *category*: contains an array of strings listing possible categories from which the message could be
(iv) *regex*: a string representation of the regular expression that will be used to match the message body, and
(v) *rate*: a dataset consisting of two numbers, indicting the *occurrence* and *timespan*, that specifies the minimum occurrence rate of a message before it flags the fact as true.

The DCM heart attack fact in section 2.2 can be expressed as in table 1:

Table 1. Configuration block for DCM heart attack fact

```
fact_dcm_heartattack : {
    description : "DCM fails to respond to a heartbeat check"
    severity : WARNING
    source : [ "RunControl" ]
    category : [ "RegularCheck" ]
    regex : ".*DCM.*missed a heartbeat"
    rate : { occurrence : 10  timespan : 60 }
}
```

Rules are configured in a very similar way, except that a rule dataset only contains two fields and an optional description. The *condition* is a string representing the Boolean expression made up of facts using the *AND* operator ("&&"), and the *OR* operator ("||"). Parentheses ("()") are also supported to override the precedence of operations. The *action* is a dataset describing the actions that need to be taken after the condition has been evaluated to be true. The rule in section 2.3 can be written as follows:

Table 2. Configuration block for DCM heart attack rule

```
rule_dcm_heartattack : {
    description : "DCM heart attack"
    condition : "fact_dcm_heartattack && fact_during_data_taking"
    action : {
        email : {
            recipient: "nova_shifter@fnal.gov"
            subject: "DCM in trouble"
        }
    }
}
```

3. Message Analyzer with extended facts and rules

A practical problem was found when implementation began for the Message Analyzer model discussed in section 2. As mentioned in the introduction, NOνA DAQ has several groups of components that are functionally similar, but are yet distinguishable. This includes approximately 200 DCMs, and a similar number of Buffer Farm nodes. When composing a fact or a rule that is suitable for a group of nodes, such as“lost connection with buffer node n”, it is not practical to write 200 separate facts or rules for every single buffer node in the system. Even worse, when talking about a rule like “dcm x cannot talk to buffer node y,” it would need $200 \times 200 = 40,000$ rules to cover all the possibilities! Grouping components with similar functionality is common in most systems. In this section we discuss the approach used to solving this problem.

3.1. Extended Message Analyzer Facts

Log message are issued by components in a running system, which can be regarded as the *source* of a message. When a node reports status through log messages, the target of a message usually falls into two situations, either the message is about the issuer itself, such as “DCM05” says “detected a memory overflow”; or the message is about a component that the issuer directly interacts with, such as “DCM05” reports “cannot communicate with buffer node 03”. In the first case, the *target* of the message is itself, and in the second case the target is “buffer node 03”. When interpreting facts from log messages, *source* and *target* are important properties, and can be factorized from a message with relatively small effort.

The simplest way of generalizing a fact to cover a set of similar facts is to use wildcards or regular expression (regex) patterns, to match specific fields of a message. Viable places for using the wildcards are for identification of the issuer of the message (source of the message), and/or within the message body (could semantically contain the target of the message). For example, one may specify a regex pattern “DCM.*” in the issuer field of a fact configuration block, and use “communication failure with (buffer node $\dot{*}$)” in the message body pattern to capture facts where any components with a prefix of “DCM” cannot talk to a buffer node in the system. The alternative would be writing a list of fact rules for each possible pair of DCM and buffer nodes in the system.

However, this simple treatment causes problems in error detection rules. For instance, a rule (R_{bnf}) of detecting a failed buffer node is built based upon an AND-relation of two facts:

$$F_1 := \text{“dcm cannot talk to a buffer node”} \tag{1}$$

$$F_2 := \text{“RunControl reports that a buffer node misses heart beats”}, \tag{2}$$

and

$$R_{bnf} := F_1 \text{ AND } F_2. \tag{3}$$

F_1 is flagged as true when a message arrives saying "dcm cannot talk to buffer node 05", and F_2 gets latched after RunControl reports "buffer node 03 missed heart beats". These two are independent facts, but they are going to trigger a false alarm about one buffer node that failed.

The problem is eliminated when the outcome flag of a generalized fact gets extended to be an array of flags. It is one dimension if only the source or the target of a fact has been generalized. It is two dimensions if both the source and the target of a fact have been generalized. For this purpose, an additional data field has been added to the fact configuration block indicating the generality of the the fact,

```
fact_name : {
    ...
    granularity : {
        per_source : bool
        per_target: bool
        target_group: int
    }
}
```

Here *per_source* and *per_target* are used to specify whether the fact is generalized in sources and/or in targets. Since the target of a message is extracted from the message body through a regex pattern, it is also required to indicate which regex group (*target_group*) is to be interpreted as the target field by the message analyzer.

3.2. Extended Rules

In the condition part of an event detection rule, the Boolean expression is evaluated based on facts. Differing from a simple fact, an extended fact represents a collection of simple facts, each of which has an individual Boolean flag. Given a rule consisting of m facts, each fact F_i $(i \leq m)$ has a degeneracy of $n_i = S_{F_i} \times T_{F_i}$, where S_{F_i} and T_{F_i} are the number of possible sources and targets of fact F_i. The rule needs to exhaustively search through $\prod_{i=1}^{m} n_i$ possible fact states to find a combination that holds for the condition of the rule. We call these fact states the *fact domain space* of a rule.

Often in the real world, the fact domain space is largely restricted. More importantly, these restrictions are necessary for a rule to achieve what it is designed for. Following the discussion of the "buffer node failure" rule example (eq 3 from section 3.1), a sensible rule using extended facts does not only need to be aware of the separation of sources and targets in the facts, but also requires a restriction. This restriction may be stated literary as

$$\text{"where the target of } F_1 \text{ is the same as the target of } F_2\text{".} \tag{4}$$

We call the above statement the *restriction clause* of a rule. An *extended rule* is a basic rule with an optional restriction clause.

3.2.1. Restriction clauses The *restriction clause* not only significantly reduces the fact domain spaces when searching for possible combinations for the condition to be held, but also creates strong bonds among individual facts, allowing them to work together. We provide two types of restriction lexemes. These lexemes can be used for building complex restriction clauses. The first one is:

$$\text{the } \textit{source} \text{ or } \textit{target} \text{ of fact } F_a \text{ is/contains a } \textit{"string literal"}. \tag{5}$$

The other one is:

$$\text{the } \textit{source} \text{ or } \textit{target} \text{ of fact } F_a \text{ is the same as the } \textit{source} \text{ or } \textit{target} \text{ of fact } F_b\ . \tag{6}$$

Lexeme (6) can be further extended to contain more than two facts to form a chain. At the end of the chain, an optional string literal can be attached in order to apply a strong restriction to every fact in the chain. This is equivalent to a set of Lexeme (6) and an optional Lexeme (5) connected by "and"s between any of them. The extended lexeme (6) is as follows.

$$\begin{gathered}\text{the } \textit{source} \text{ or } \textit{target} \text{ of fact } F_a \text{ is the same as the } \textit{source} \text{ or } \textit{target} \text{ of fact } F_b,\\ \text{and is the same as the } \textit{source} \text{ or } \textit{target} \text{ of fact } F_c, \text{...},\\ \text{optionally, all of the fields must equal to a "string literal"}\end{gathered}$$

Complex restriction clauses are built on Lexeme (5) and (6), or its extended form. The outcome of a lexeme is a subdomain of fact spaces, therefore it is logical to use the *union* and *intersection* operators while connecting any of two lexemes. Also note, that since the complete space of a fact is often undefined, the complement operation or lexeme therefore cannot be supported in this context.

Evaluating the condition of a rule, as well as the domain operations the application of restriction clauses for a rule are largely analogous to concepts in relational databases. Each fact can be considered as a database table with sources and targets as the rows and columns. For a rule without a restriction clause, the fact domain space forms a join table from all facts. The evaluation process then consists of searching through the joined table for a value set that satisfies the Boolean expression of the condition. Restriction clauses are equivalent to the conditional join of tables.

3.2.2. Formalizing with a domain specific language It is necessary to provide a well defined and structured language to ease describing the conditional part of a rule, as well as defining the restriction clauses. A domain specific language has been engineered for this purposes.

First of all, a fact is referenced by its name, which forms the conventional naming rules of an identifier. Fields of a fact can be accessed by the point operator followed by a dollar sign, along with a predefined short name for the fields. Specifically, the source of a fact is denoted by ".$\$s$", and the target is by ".$\$t$". We use "&&" and "||" for *AND* and *OR* operators within Boolean expressions. A restriction clause starts with the keyword *"where"*. *Unions* and *intersections* are denoted by "OR" and "AND". The expression is evaluated from left to right and in a higher-to-lower order of precedence ("&&" > "||", "AND">"OR"). It can also be overridden by using parentheses. Following the language specification, the rule described in eq. (3), as well as its restriction clause eq. (4), can be formalized and expressed as follows:

$$R_{bnf} := F_1 \ \&\& \ F_2 \text{ where } F_1.\$t = F_2.\$t. \tag{7}$$

4. Generic message analyzer with user-defined functions

The Message Analyzer, with its support for extended facts and rules, is able to cover a majority of the use cases defined for finding runtime abnormalities for a complex system. However, we have found cases where a fact or an event cannot be easily extracted using message filtering or regex pattern recognition. In this section, we follow examples to introduce the feature of user-defined functions that enable the Message Analyzer to handle a great deal of more complex situations using customized logic implemented in generic programming language such as C++.

4.1. User-defined test function for facts

Fact extraction rules provide five tests against incoming messages. To assert status using log message, these rules include the issuer, severity, category, regex match, and occurrence rate as discussed in section 2.2. In some cases, however, facts are extracted not only by matching a predefined pattern, but also by some further interpretation of the message. As an example, a sensor periodically sends back temperature readings through log messages. Our desire is to have a fact asserted when the temperature reading is above some pre-configured threshold in several consecutive messages. This can be achieved by carefully composing a regex pattern. However, it is more convenient if the Message Analyzer understands the temperature as a number and can compare it directly with a preset value.

For this purpose, we have designed a plugin interface with necessary APIs for the Message Analyzer. The interface allows users to implement custom logic in C++ in the form of a user-defined function. The function can be referenced in the configuration file and invoked as preconditions are met. When needed, it is also permitted for users to define arguments for the function (e.g., the threshold temperature used for comparison). The built-in parser will recognize strings, numbers, and booleans as arguments and have them passed to the function when it invoked.

User-defined functions are loaded and invoked dynamically while the Message Analyzer is running. A customized test field with basic Boolean logic has been added to the configuration block to utilize custom functions, expressed as follows:

```
fact_name : {
    ...
    test : "test expression"
}
```

Customized test functions and basic arithmetic functions can be included in the *test expression.* The formal grammar represented in EBNF is shown as in table 3.

Table 3. EBNF for *test expressions*

```
test_expr        := boolean_and_expr ( ‘||’ boolean_and_expr )*
boolean_and_expr := boolean_primary ( ‘&&’ boolean_primary )*
boolean_primary  := function | ‘(’ test_expr ‘)’
function         := function_name ‘(’ [  value ( ‘,’ value )* ] ‘)’
                    [ compare_op value ]
function_name    := alpha_lead_string
compare_op       := ‘<’ | ‘<=’ | ‘==’ | ‘!=’ | ‘>=’ | ‘>’
value            := string | bool | float
```

4.2. User-defined condition function for rules

Similar to user-defined test functions for facts, we have also introduced user-defined *condition functions* for event extraction rules. Consider a situation in which one of the buffer nodes has failed. Within seconds following the failure of the buffer node, we will receive a list of status messages from a set of DCMs, stating they have failed to send data to a certain buffer node. A fact has been set to trap such log messages. However, we would like to hold off producing an alarm until a sufficient fraction of DCMs has reported the failure to the same buffer node. This is done in order to suppress possible false alarms. In this use case, we would like to have a user-define function that operates on the fact to figure out how many individual sources have been asserted due to it.

Composing a user-defined condition function, and registering the function to the Message Analyzer is very similar to what has to be done for a user-defined test function. However, we want to point out a difference between the two types of functions. A fact test function is applied to a sequence of log messages that pass the previous checks. A rule condition function is applied on facts. Therefore, a fact test function usually has access to the input messages, while a rule condition function may contain knowledge regarding several facts and their related messages.

The complete EBNF for the condition portion of an event extraction rule after introduction of the user-defined condition functions is shown in table 4.

Table 4. EBNF for the *condition expression* of an event extraction rule

```
cond_expr          := boolean_expr [ ‘WHERE’ domain_expr ]

boolean_expr       := boolean_and_expr ( ‘||’ boolean_and_expr )*
boolean_and_expr   := boolean_primary  ( ‘&&’ boolean_primary  )*
boolean_primay     := fact | function | ‘(’ boolean_expr ‘)’

domain_expr        := domain_and_expr ( ‘OR’  domain_and_expr )*
domain_and_expr    := domain_primary  ( ‘AND’ domain_primary  )*
domain_primary     := fact ‘.$’ ‘s’|‘t’ ( ‘=’ fact ‘.$’ ‘s’|‘t’ )*
                      [ ‘=’ ‘ANY’ | string ]
                      | ‘(’ domain_expr ’)’

function           := fun_name ‘(’ fact [ ‘.$’ ‘s’|‘t’ ] [ ( ‘,’ value )* ] ‘)’
                      [ compare_op value ]

fact               := alpha_lead_str
fun_name           := alpha_lead_str
value              := string | bool | float

compare_op         := ‘<’ | ‘<=’ | ‘==’ | ‘!=’ | ‘>=’ | ‘>’

string             := alpha_lead_str
alpha_lead_str     := ‘_a-zA-Z’ ( ‘_a-zA-Z0-9’ )*

bool               := ‘true’ | ‘false’
```

5. Conclusions

In this paper we have discussed the design of a status message correlation analysis tool for capturing abnormalities and other events at system scale. We have abstracted the analysis into two layers of cascading reasoning: a fact extraction layer based on status messages, and an event extraction layer based on facts. Fact and event extraction rules are further extended to support the grouping of similar components without losing resolution when identifying the problematic part of the system. User-defined functions have added more flexibility to this analysis tool by enabling custom logic to recognize facts and events using generic programming language such as C++. Finally, we described a domain specific language for expressing the fact and event extraction rules in an efficient and user-friendly way.

The Message Analyzer has been deployed at the NOνA Near Detector On Surface (NDOS) for many months, and has since become a valuable tool for DAQ monitoring and assurance of data quality. It also helps the operations crew diagnose DAQ problems. Currently we are in the phase of collecting more rules and experience during daily operation. For this purpose, in addition to the live message streams that the Message Analyzer currently relies on, we have also added a message player that can play back archived messages from files or permanent storage for the software to perform analysis and experimenting new rules based on historical events. As the NOνA far detector commissioning deadline at Ash River gets close, the next step will be to deploy the Message Analyzer, along with the knowledge bases, to assist in the DAQ operation. The tools can also be used to move towards automated operations, including the automatic detection of and recovery from non-fatal errors without human interventions.

Acknowledgments

The authors wish to express their gratitude to Dr. Marc Paterno, Dr. Peter Shanahan for all the inspiring discussions on the design of the Message Analyzer package. We would also like to convey thanks to Dr. Peter Shanahan and all NOνA online DAQ software team fellows for providing us with numerous use cases; for insightful feedback from field tests and daily use of the Message Analyzer package in the NOνA DAQ system.

References

[1] Nowak J and NOνA Collaboration 2012 *Status of the NOvA Experiment*, *AIP Conf. Proc.* **1441** 423.
[2] Kasahara S M 2011 *NOνA Data Acquisition System*, *Proc. of the Technical Instrumentation in Particle, Physics Conference* (in publication).
[3] `http://ganglia.sourceforge.net`, retrieved on 06/21/2012.
[4] Lu Q, Kowalkowski J B and Biery K A 2011 *J. Phys.: Conf. Series* **331** 022017.
[5] `https://cdcvs.fnal.gov/redmine/projects/messagefacility`, retrieved on 06/21/2012.
[6] Leondes C T 2002 *Expert systems: the technology of knowledge management and decision making for the 21st century*, pp 1–22.
[7] Barzilay R, McCullough D, Rambow O, DeChristofaro J, Korelsky T and Lavoie B 1998 *A new approach to expert system explanations. In 9th International Workshop on Natural Language Generation* pp 78–87.
[8] `http://en.wikipedia.org/wiki/Forward_chaining`, retrieved on 06/21/2012.
[9] Dittrich K, Gatziu S and Geppert A 1995 *The Active Database Management System Manifesto: A Rulebase of ADBMS Features. Lecture Notes in Computer Science 985* (Springer) pp 3–20.
[10] Paton N 1999 *Active Rules in Database Systems* (Springer-Verlag).
[11] Widom J and Ceri S 1995 *Active Database Systems* (San Mateo: Morgan-Kaufmann).
[12] Adi A, Botzer D, Etzion O and Yatzkar-Haham T 2000 *Push technology personalization through event correlation*, *Proc. 26th Int. Conf. on Very Large Databases* pp 643–5.
[13] Bonifati A, Ceri S and Paraboschi S 2001 *Active rules for XML: A new paradigm for e-services*, *VLDB J.* **10** 1 pp 39–47.
[14] Abiteboul S, Vianu V, Fordham B S and Yesha Y 2000 *Relational transducers for electronic commerce*, *JCSS* **61** 2 pp 236–69.
[15] `http://www.prismtech.com/opensplice/products/opensplice-dds-overview`, Retrieved on 06/12/2012.
[16] `https://cdcvs.fnal.gov/redmine/projects/fhicl`, retrieved on 06/21/2012.