CHEP 2012, New York

Qiming Lu
Scientific Software Infrastructure Group

Message Analyzer for NOvA

Fermilab
Fermi National Accelerator Laboratory
Office of Science / U.S. Department of Energy
Managed by Fermi Research Alliance, LLC

Automatic Error Recovery System for NOvA

- To maintain a healthy running state for a complex system with a large number of components, such as NOvA DAQ, mechanics are needed for
  1. continuously monitoring status of each participating component
  2. detect and react to events that might put the system or the quality of data in jeopardy with minimum delay
- Automatic Error Recovery System (AERS) for NOvA Online DAQ
  - Status reports from each component, and from dedicating monitoring nodes
  - Decision making engine performs correlation analysis based on status reports
  - Error Handling Supervisor reacts to situations and carries out actions

AERS: *Status Report*

- Reports from each component regarding its own status and sometimes the status of its direct interacting components
- Reports from dedicated monitoring nodes (e.g., Ganglia) regarding general status and customized matrixes
- Reports from centralized RunControl / AppManager, etc. regarding the run state and other info
- Status reports are issued, routed, collected and stored through a distributed message service package called ***MessageFacility***

** **MessageFacility** is developed at SSI group@FNAL as a generic messaging package used by art Framework and other experiments*

AERS: *Message Analyzer*

- Collects status messages in the system and performs correlation analysis for event/error identification
- Semantic reasoner to infer logical consequences from a set of asserted facts
- In particular, we are interested in the rule engine that processes ***Event-Condition-Action*** (ECA) rules
  - ECA is a widely used model in inference rule engines with a general syntax of

on ***Event*** if ***Condition*** then ***Action***

AERS: *Error Handling Supervisor*

- The action part of an ECA rule submits the detected error to the supervisor
- Supervisor takes proper action based on errors submitted, and based on its knowledge about this error
- Might have multiple instances of Message Analyzer but only one instance of Error Handling Supervisor is allowed for each partition

Overview of Message Analyzer

- Real-time event correlation analysis tool based on log messages
- Implements a forward chaining inference engine
- Deals with two levels of cascading ECA rules:
  - ***Fact extraction***: driven by incoming messages to extract facts from status logs
  - ***Event identification***: correlation analysis based on facts to identify events
  - ***Cascading rules***: when an rule is triggered for querying its condition on an incoming event, executing its action may in turn trigger new rules for further assessment
- Flexible and user-configurable

Message Analyzer: *Fact Extraction*

- Facts are assertions based on observations from log messages:
  - ***Logs:*** *"Sensor 7 reads 101 °F", "Sensor 7 reads 107 °F", "Sensor 7 reads 104 °F"*

    ⇨ ***Fact:*** *"Sensor 7 reads high temperature"*
- Facts are extracted by performing a series of tests on incoming messages
  - Tests on the *issuer*, *severity* and *category* of a message
  - Test whether the *message body* matches a regex pattern
  - Test whether the occurrence rate of the message exceeds some preset threshold
- The output of a fact extracting rule is a ***Boolean flag*** indicating whether the related fact has happened in the system or not

Message Analyzer: *An Example Fact*

- *During data taking, NOvA RunControl sends out a heartbeat check every other second to all components in the system, and expects responses from all of them. Otherwise emits a status message.*

- Fact "*DCM has a heart attack*" is asserted through following tests on status messages:
  - Test whether the sender is "*RunControl*"
  - Test whether the severity is "*Warning*" or higher
  - Test whether the category is "*RegularCheck*"
  - Test whether the body matches regex expression "*DCM missed heartbeats*"
  - Test whether the message has been seen over *10 times* in the past *60 seconds*

- If passes all five tests, a flag of ***TRUE*** is marked under the fact "*DCM has a heart attack*"

Message Analyzer: *Event Identification Rules*

- Identifying events based on single facts
  - "event of ***dcm_failure*** upon detection of ***dcm_heartattack***"
  - Often times a single fact is not sufficient to identify the exact location of a problem;
  - Or we might need multiple sources to confirm a situation
- So for better capturing an event,

Message Analyzer: *Event Identification Rules*

- ***Use logically correlated facts to identify events***
  - "event of ***dcm_failure*** is asserted when both ***dcm_heartattack*** and ***dcm_selfcheck_failure*** flagged during ***data_taking*** or ***hardware_config*** stage"
  - In a formal language,

    dcm_failure ***:=***
    dcm_heartattack && dcm_selfcheck_failure &&
    (data_taking || hardware_config)

- The condition part of an event identification rule is a Boolean expression made up of facts
- An event identification rule is called a ***rule***

Message Analyzer for NOvA: *Problem*

- NOvA far detector consists of around *180* DCMs, *180* buffer nodes, and a handful of loggers and other nodes, etc.
- Write *180* ***dcm_heartattack*** facts which are only slightly different from each other seems not very realistic…
- Even worse, if we ever need a fact of ***dcm_bn_conn_failure***, will we need to provide *180 * 180 = 32,400* separated fact extraction rules?

Message Analyzer for NOvA: *Approach*

- Use either ***wildcards***, or ***regular expressions*** to collapse similar facts
- Facts:
  - ***DCM_Heartattack :=***

    RunControl : "dcm-?? missed heartbeats"
  - ***DCM_BN_Conn_failure :=***

    dcm-?? : "cannot connect with bn-??"

Message Analyzer for NOvA: *New Problems*

- Facts:

  F1 := RunControl: dcm-?? missed heartbeats

- ***Problem 1: Frequency tests get messed up***

  *RunControl: **dcm-07** missed heartbeats*
  *RunControl: **dcm-03** missed heartbeats*
  *RunControl: **dcm-02** missed heartbeats*
  *RunControl: **dcm-07** missed heartbeats*
  …

Message Analyzer for NOvA: *New Problems*

- Facts:

  F1 := RunControl: dcm-?? missed heartbeats
  F2 := dcm-??: I'm not happy

- Rule:

  R1 := F1 && F2

- ***Problem 2: Does not work with correlated events:***

  *RunControl: "**dcm-07** missed heartbeats"*
  ***dcm-02**: "I'm not happy"*

Improved Approach: *Extended Facts*

- Every log message has a ***source***, which is the issuer of the message
- Semantically, most messages are likely to have a ***target***
  - If the message is about the sender itself, the target is itself:
    "*dcm-05*" says "failed to start data flow"
  - Or the message talks about a direct interacting component of the issuer
    "*dcm-05*" says "cannot connect to *buffer node 03*"
- Only apply wildcards to the source and/or target field of a message
- Extend a primitive fact into a collection of related facts by making distinguish of the ***sources*** and ***targets*** of a message

Extended Facts: *Outputs*

- *RunControl: "dcm-?? missed heartbeats"*
  - *Source: RunControl* *(# of sources = 1)*
  - *Target: a collection of dcms* *(# of targets = n)*
- *Dcm-?? : "failed to start data flow"*
  - *Source: a collection of dcms* *(# of sources = n)*
  - *Target: self* *(# of targets = 1)*
- *Buffernode-?? : "cannot connect to dcm-??"*
  - *Source: a collection of buffernodes* *(# of sources = m)*
  - *Target: a collection of dcms* *(# of targets = n)*

- Output of a ***primitive fact***: one single Boolean value
- Output of an ***extended fact***: an array of Boolean values in either one dimension or two dimensions

Extended Rules in Coping with Extended Facts

- Facts:

  F1 := RC: dcm-?? missed heartbeats
  F2 := dcm-??: I'm not happy

- Rule:

  R1 := F1 && F2

- To make the rule complete: add a ***restriction clause*** to the rule,

  R1 := F1 && F2 where the target of F1 is the same as the source of F2

- *How to compose a restriction clause?*

Extended Rules: *Restriction Clauses*

- Restriction lexeme/token 1:
  - *the source/target of fact F1 is a "string literal"*
  - ***F1.$(s|t) = "string literal"***
- Restriction lexeme/token 2:
  - *The source/target of fact F1 is the same as the source/target of fact F2*
  - ***F1.$(s|t) = F2.$(s|t)***
- Chained form of restriction lexeme 2 (to save some typing):
  - ***F1.$(s|t) = F2.$(s|t) = F3.$(s|t) = … = "string literal"***
- Complex restriction clauses are built on lexeme 1 and 2:
  - ***F1.$s = F2.$t AND (F2.$s = "DCM" OR F2.$s = "buffer node")***

Extended Rule In *A Formal Language*

- Facts:

  F1 := RC: dcm-?? missed heartbeats
  F2 := dcm-??: I'm not happy

- Rule with a restriction clause:

  R1 := F1 && F2 where the target of F1 is the same as the source of F2

- In the formal language

  R1 := F1 && F2 where F1.$t = F2.$s

- *Have completed EBNF and parser for the rule description language*

Analogy of extended rules with relational databases

- *Domain space of an extended fact*
  ⇔ (analogies to) *database table*:
  - A collection of facts distinguished by source/target ⇔ a collection of values in a database table in rows/columns

- *Evaluating a rule's condition* ***without*** *a restriction clause*
  ⇔ searching through the table which is ***joined*** from all fact tables for combinations that holds the condition

- *Evaluating a rule's condition* ***with*** *a restriction clause*
  ⇔ searching through the table ***conditionally joined*** from all fact tables for combinations that holds the conditions

Facts and Rules with *User-defined Functions*

- *There are cases that a fact or an event cannot be simply extracted through generic message filtering or regex pattern recognition*
- Example 1: a sensor sends back temperature readings every minute
  - We are interested when *the last five readings are higher than a preset threshold*
- Example 2: a collection of detectors periodically send back synchronization info
  - We are interested whether *all the detectors are in-sync (all synchronized to the same time)*
- Adding ***user-defined functions*** to do some interpretations or simple logic/arithmetic to the message would add great flexibilities to the system

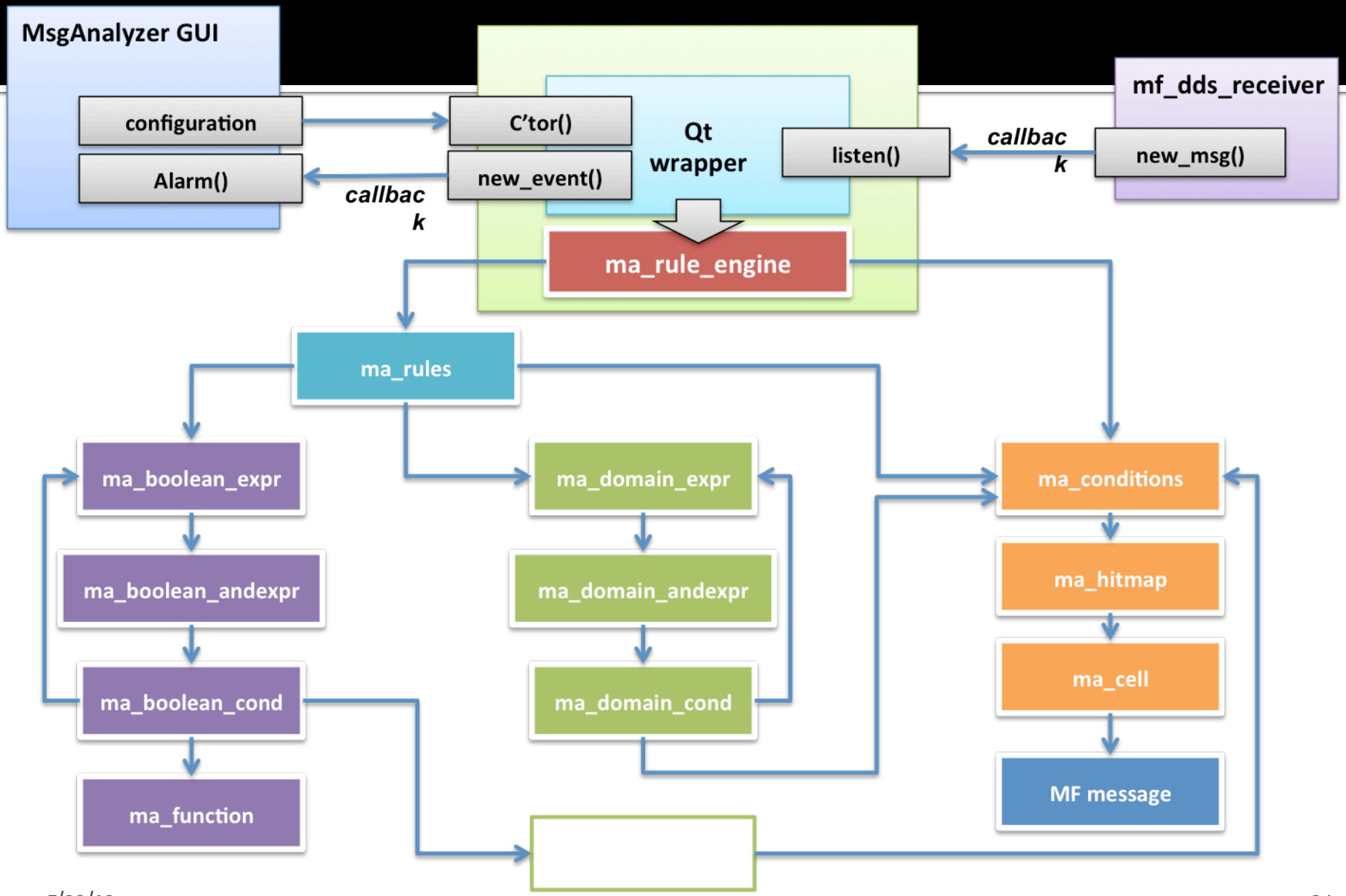
What Can User-defined Functions Do?

- ***Customized logic written in C++***
- Having access to all related fields of the latest message, and all previous messages trapped in the fact
- User functions are implemented by C++ classes – *they can have memories*
- *Can have arguments to the function*
- Loaded and invoked dynamically during the runtime

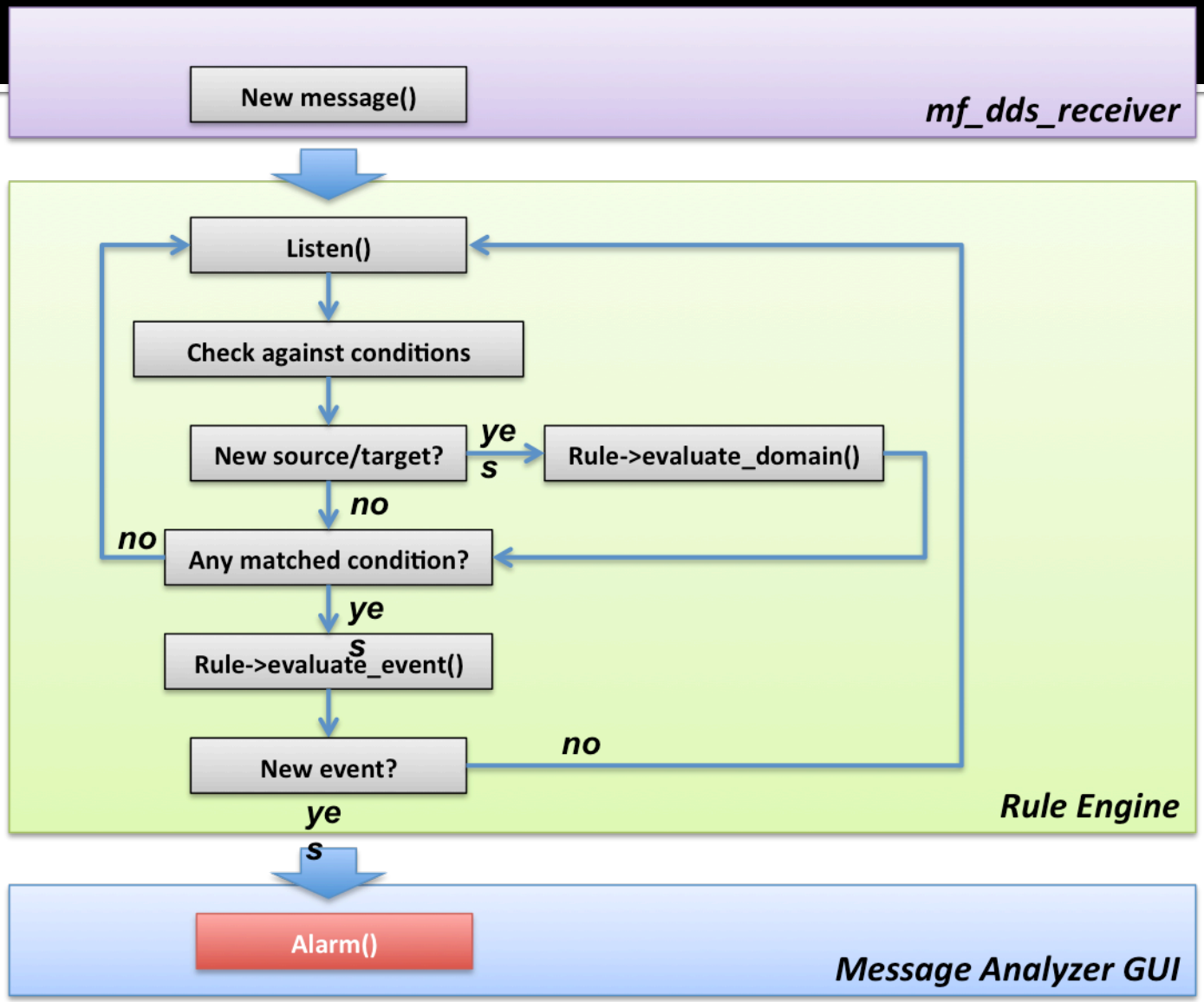
Summaries

- ***Message Analyzer*** is a light-weight correlation analysis tool with great flexibility and extensibility
- It has been in use for monitoring the run state and data quality in the *NOvA NDOS DAQ* system for months
  - With some exciting results
  - In the process of inventing new rules through the study of historical logs of real events
- Separation of the system knowledge (the rules) from the software implementation
  - Composing facts/rules using a *Domain-specific Rule Language*
- The package has been made generic and portable
  - Easy migration to other experiments/systems

Message Analyzer: *Class Diagrams*



Message Analyzer: Execution Flow Chart



User-defined Functions: *An Example*

- A sensor sends back temperature readings every minute
  - We would like fact "***high_temp_reading***" to be flagged when the last five readings from that sensor are higher than a preset threshold
  - Can be achieved by playing with the regex pattern
  - However, it would be great if we could simply do this in the configuration block of a fact:

    regex : "temperature reading (\d*)"
    custom_test : "string_to_number(1) > 100"

  - *string_to_number()* is a **user-defined function** that parses a string to a number
  - Parameter to the function is a regex group index to extract the interesting part from the message body

User-defined Function: *More Examples*

- User-defined function for fact extraction rules:
  - str_to_number(int) – # parse the string in given regex group to a number
  - is_synchronized(float error) – # selected components are synchronized within +/-error

- User-defined functions for event identification rules:
  - Count(Fact f, string type) -- # count the number of sender/receiver (specified by type) in the fact f
  - Count_percent(Fact f, string type) -- # count the percentage of sender/receiver (specified by type) in the fact f

Extended Condition: Introducing Domains

- The ***domain of definition***, or simply the ***domain*** of a condition is the set of *(source, target)* pairs for which the condition is defined. That is, the condition provides an output (a Boolean) for each pair of the domain

- Domain of a primitive condition

	Source = NULL
Target = NULL	Boolean output O

- Domain of an extended condition in either source or targets

	S1	S2	S3	S4
Target	O(1,1)	O(2,1)	O(3,1)	O(4,1)

	Source
T1	O(1,1)
T2	O(1,2)
T3	O(1,3)
T4	O(1,4)

Extended Condition: Introducing Domains

- The ***domain of definition***, or simply the ***domain*** of a condition is the set of *(source, target)* pairs for which the condition is defined. That is, the condition provides an output (a Boolean) for each pair of the domain

- Domain of an extended condition in both source and targets

	S_1	S_2	S_3	S_4
T_1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
T_2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
T_3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
T_4	O(1,4)	O(2,4)	O(3,4)	O(4,4)

Domain Operations: Subset of A Domain

- The source of a condition C1 is "S2"

*In **C1.$s**, **C1** is the condition name, **$s** means the source of the condition; Similarly we can have **$t** representing the target of a condition*

C1.$s = "S2"

	S1	S2	S3	S4
T1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
T2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
T3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
T4	O(1,4)	O(2,4)	O(3,4)	O(4,4)

(2, *)

In (source, target) form, where source and target are 1-based index, '' represent any*

Domain Operations: Correlation Between Domains

- Condition *C1* and *C2* each populated with some amount of messages

Sources

Targets

	dcm01	dcm02	dcm03	dcm04
bn01	O(1,1)	O(2,1)	O(3,1)	O(4,1)
bn02	O(1,2)	O(2,2)	O(3,2)	O(4,2)
bn03	O(1,3)	O(2,3)	O(3,3)	O(4,3)
bn05	O(1,4)	O(2,4)	O(3,4)	O(4,4)

Condition C1

Sources

Targets

	bn01	bn03	bn04	bn12
logger1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
logger2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
logger3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
logger 4	O(1,4)	O(2,4)	O(3,4)	O(4,4)

Condition C2

Domain Operations: Union and intersection

- The source of a condition C1 is either "dcm02", ***OR*** "dcm04"

C1.$s = 'dcm02' OR C1.$s = 'dcm04'

	dcm01	dcm02	dcm03	dcm04
bno1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
bno2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
bno3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
bno5	O(1,4)	O(2,4)	O(3,4)	O(4,4)

Domains	Condition C1
1	(2, *)
2	(4, *)

Domain Operations: Union and intersection

- The source of a condition C1 is "dcm03", ***AND*** the target of C1 is "bno2"

C1.$s = 'dcm03' AND C1.$t = 'bno2'

	dcm01	dcm02	dcm03	dcm04
bno1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
bno2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
bno3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
bno5	O(1,4)	O(2,4)	O(3,4)	O(4,4)

Domains	Condition C1
1	(3, *) ∧ (*, 2) = (3, 2)

Domain Operations: Correlation Between Domains

- The target of condition C1 is the same as the source of condition C2

C1.$t = C2.$s

	dcm01	dcm02	dcm03	dcm04
bno1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
bno2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
bno3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
bno5	O(1,4)	O(2,4)	O(3,4)	O(4,4)

	bno1	bno3	bno4	bn12
logger1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
logger2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
logger3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
logger 4	O(1,4)	O(2,4)	O(3,4)	O(4,4)

Domains	Condition C1	Condition C2
1	(*, 1)	(1, *)
2	(*, 3)	(2, *)

Domain Operations: Correlation Between Domains

- The target of condition C1 is the same as the source of condition C2, and they are both 'bno3'

C1.$t = C2.$s = 'bno3'

	dcm01	dcm02	dcm03	dcm04
bno1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
bno2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
bno3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
bno5	O(1,4)	O(2,4)	O(3,4)	O(4,4)

	bno1	bno3	bno4	bn12
logger1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
logger2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
logger3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
logger 4	O(1,4)	O(2,4)	O(3,4)	O(4,4)

Domains	Condition C1	Condition C2
1	(*, 3)	(2, *)

Domain Operations: Combined Ops – Case 1

C1.$t = C2.$s OR C1.$t = 'bno3'

Step1: ***C1.$t = C2.$s OR C1.$t = 'bno3'***

	dcmo1	dcmo2	dcmo3	dcmo4
bno1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
bno2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
bno3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
bno5	O(1,4)	O(2,4)	O(3,4)	O(4,4)

	bno1	bno3	bno4	bn12
logger1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
logger2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
logger3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
logger 4	O(1,4)	O(2,4)	O(3,4)	O(4,4)

Domains	Condition C1	Condition C2
1	(*, 1)	(1, *)
2	(*, 3)	(2, *)

Domain Operations: Combined Ops – Case 1

C1.$t = C2.$s OR C1.$t = 'bno3'

Step2: *C1.$t = C2.$s OR* ***C1.$t = 'bno3'***

	dcm01	dcm02	dcm03	dcm04
bno1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
bno2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
bno3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
bno5	O(1,4)	O(2,4)	O(3,4)	O(4,4)

	bno1	bno3	bno4	bn12
logger1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
logger2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
logger3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
logger 4	O(1,4)	O(2,4)	O(3,4)	O(4,4)

Domains	Condition C1	Condition C2
1	(*, 3)	(*, *)

Domain Operations: Combined Ops – Case 1

C1.$t = C2.$s OR C1.$t = 'bno3'

Step2: *C1.$t = C2.$s OR C1.$t = 'bno3'*

V

Domains	Condition C1	Condition C2
1	(*, 1)	(1, *)
2	(*, 3)	(2, *)

Domains	Condition C1	Condition C2
1	(*, 3)	(*, *)

check pairwise inclusion

Domains	Condition C1	Condition C2
1	(*, 1)	(1, *)
2	(*, 3)	(*, *)

Domain Operations: Combined Ops – Case 2

C1.$t = C2.$s AND C1.$t = 'bno3'

Step1: ***C1.$t = C2.$s*** *AND C1.$t = 'bno3'*

	dcmo1	dcmo2	dcmo3	dcmo4
bno1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
bno2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
bno3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
bno5	O(1,4)	O(2,4)	O(3,4)	O(4,4)

	bno1	bno3	bno4	bn12
logger1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
logger2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
logger3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
logger 4	O(1,4)	O(2,4)	O(3,4)	O(4,4)

Domains	Condition C1	Condition C2
1	(*, 1)	(1, *)
2	(*, 3)	(2, *)

Domain Operations: Combined Ops – Case 2

C1.$t = C2.$s AND C1.$t = 'bno3'

Step2: *C1.$t = C2.$s AND* ***C1.$t = 'bno3'***

	dcm01	dcm02	dcm03	dcm04
bno1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
bno2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
bno3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
bno5	O(1,4)	O(2,4)	O(3,4)	O(4,4)

	bno1	bno3	bno4	bn12
logger1	O(1,1)	O(2,1)	O(3,1)	O(4,1)
logger2	O(1,2)	O(2,2)	O(3,2)	O(4,2)
logger3	O(1,3)	O(2,3)	O(3,3)	O(4,3)
logger 4	O(1,4)	O(2,4)	O(3,4)	O(4,4)

Domains	Condition C1	Condition C2
1	(*, 3)	(*, *)

Domain Operations: Combined Ops – Case 2

C1.$t = C2.$s AND C1.$t = 'bno3'

Step2: *C1.$t = C2.$s AND C1.$t = 'bno3'*

Domains	Condition C1	Condition C2
1	(*, 1)	(1, *)
2	(*, 3)	(2, *)

$\wedge$

Domains	Condition C1	Condition C2
1	(*, 3)	(*, *)

check pairwise intersection

Domains	Condition C1	Condition C2
1	(*, 3)	(2, *)

Extended Rules in Coping with Extended Conditions

- Conditions:

  C1 := RC: dcm-?? missing heartbeats
  C2 := dcm-??: I'm not happy

- Rule:

R1 := C1 && C2 Given that the target of C1 is the same as the source of C2

R1 := C1 && C2 WHERE C1.$t = C2.$s

Configuration for Message Analyzer

- Conditions and rules are delivered in configuration files written in ***FHiCL***
- Skeleton of the configuration file:

```
conditions :
{
  condition_1 : { … }
  condition_2 : { … }
}

rules :
{
  rule_1 : { … }
  rule_2 : { … }
}
```

Express Conditions in Configuration Language

```
conditions :
{
  fail_conn_cond :
  {
    description : "This condition detects the conn failure between dcms and bns"
```

Filtering

```
severity : WARNING
source   : [ "dcm.*" ]
category : [ * ]
```

(...) defines regex group, can be referenced later

Matching

```
regex    : "Timeout when trying to connect with (buffernode \d\d)"
```

Freq. test

```
rate     : { occurrence:10  timespan:60 }
```

Extended Condition

```
granularity :
{
  per_source   : true  # extend in source
  per_target   : true  # extend in target
  target_group : 1     # regex group in match expression
}
```

```
  }
}
```

Example Conditions

```
Cond_1 :
{
  description : "cannot reserve resources"

  severity : ERROR
  source   : [ "runControl" ]
  category : [ * ]

  regex    : "failed to reserve resources"
}


Cond_2 :
{
  description : "dcm cannot get data flow after 10 attempts"

  severity : WARNING
  source   : [ "dcm-\\d\\d" ]

  regex    : "failed to start data flow"

  rate        : { occurrence:10 timespan:60 }
  granularity : { per_source:true }
}
```

Example Conditions

```
Cond_3 :
{
  description : “dcm cannot connect to buffer node”

  severity : WARNING
  source   : [ “dcm-\\d\\d” ]
  category : [ * ]

  regex    : “cannot connect to (buffernode-\\d\\d)”

  granularity :
  {
    per_source : true
    per_target : true
    target_group : 1
  }
}
```

Express Rules in Configuration Language

```
rules :
{
  event_dcm_heartattack :
  {
    description : "This rule detects dcm heartattack in sensible stages"

    expression : "DCM_Heartattack && DCM_SelfDiag && ( InHardwareConfig || DuringRun )
                      WHERE DCM_Heartattack.$t = DCM_SelfDiag.$s"

    message : "${DCM_SelfDiag.$s} has heart attack.
               Detailed messages: ${DCM_Heartattack.$m} \n ${DCM_SelfDiag.$m}"
  }
}
```

- Detailed grammar of the rule expression will be given in next slide
- When an event has been identified by the rule a message will be printed

Formal Grammar of Rule Condition Expression

```
rule_expr         := boolean_expr [ 'WHERE' domain_expr ]

boolean_expr      := boolean_and_expr ( '||' boolean_and_expr )*
boolean_and_expr  := boolean_primary  ( '&&' boolean_primary  )*
boolean_primay    := condition | function | '(' boolean_expr ')'

domain_expr       := domain_and_expr ( 'OR'  domain_and_expr )*
domain_and_expr   := domain_primary  ( 'AND' domain_primary  )*
domain_primary    :=   condition '.$' 's'|'t' ( '=' condition '.$' 's'|'t' )*
                                              [ '=' 'ANY' | string ]
                     | '(' domain_expr ')'

function          := fun_name '(' condition [ '.$' 's'|'t' ] [ ( ',' value )* ] ')'
                     [ compare_op value ]

condition         := 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'
```

Custom Recognition Pattern: An Example of DCM Sync

- *ma_function_is_syncd.h*

```
#ifndef ERROR_HANDLER_MA_FUNCTION_IS_SYNCD_H
#define ERROR_HANDLER_MA_FUNCTION_IS_SYNCD_H

#include <ErrorHandler/ma_function.h>

class ma_func_is_syncd : public ma_function
{

public:

  ma_func_is_syncd() : sync_time() { }
  virtual ~ma_func_is_syncd() { }

  virtual boost::any
    evaluate ( ma_condition const & cond, arg_t, ma_cond_domain );

private:

  std::string sync_time;

};

#endif
```

Custom Recognition Pattern: An Example of DCM Sync

- *ma_function_is_syncd.cpp*

```
#include <ErrorHandler/ma_function_is_syncd.h>
#include <ErrorHandler/ma_condition.h>

// class registration
REG_MA_FUNCTION( is_syncd /*function name*/, ma_func_is_syncd  /*class name*/ )

boost::any
  ma_func_is_syncd::evaluate( ma_condition const & cond, arg_t, ma_cond_domain )
{
  std::string sync = cond.get_msg_group(1);

  if( sync_time.empty() )
  {
    sync_time = sync;
    return boost::any(true);
  }
  else
  {
    return boost::any(sync != sync_time);
  }
}
```

Custom Recognition Pattern: An Example of DCM Sync

- *ma_function.h*

```
typedef std::vector<boost::any>  anys_t;

class ma_function
{
public:

  ma_function() {}
  virtual ~ma_function() {}

  virtual boost::any
    evaluate( ma_condition const & cond
            , arg_t cond_arg
            , ma_cond_domain dom ) = 0;

  virtual bool
    parse_arguments( anys_t const & args ) { return true; }

private:

};
```

Message Analyzer: Execution Flow Chart

- Deduct domains from the domain expression of the rule
- Do an exhaustive search for the event, including:
  - Take a set of Boolean values (one for each condition) from the domain, evaluate the condition expression
  - iterate to the next possible set of values from the domain for evaluation, until reach the end of the domain, or the evaluation returns true
- If the search returns true, notify the analyzer that an event has been found

Message Analyzer: Improvements

- Multi-threaded domain evaluation and event evaluation functions
- Add a buffer (producer/consumer queue) between mf_dds_receiver and the rule engine to avoid skipping of log messages
- Make an abstract layer between the message facility (or any logging libraries) and the message analyzer, so that it can use other logging facilities
- Improved event report and error recovery actions