RULE-BASED VALIDATION OF COMPLEX SYSTEM CONFIGURATIONS

Dibyendu Goswami

ABSTRACT

Real-Time Measurement Systems deployed in harsh environments are expected to behave autonomously. Before deployment the system configuration must be validated against a number of diverse requirements from various users of the system. This paper proposes a pattern to automate this validation process using a Rule Engine approach.

INTRODUCTION

There are several scenarios where real-time data acquisition is done in environments where the measurement device and the backend control and recording system are far apart and with very limited communication ability (i.e. low bandwidth and possibly intermittent connectivity and a risk of signal distortion). Examples are wireless submarine ROV operations, space operations with communication between an earth station and remote equipment, and my own domain from the oilfield which is measurement or logging while drilling operations.

Figure 1: Real-Time Measurement System Setup

Common to these systems is the value of real-time measurements made available to analysts during the operation. Not only does the real-time data provide early insights into the value of the measurements, but it may be used to revise and control the operation as it proceeds. The devices will typically record far more data than what can be
transferred real-time, so a carefully selected sub-set is set up for transmission. The core amount of data is then
dumped at the end of the operation when the device is retrieved/returned back to the same location as the back-
end system.

Because of the severe limitation on communication, the measurement device must be as independent of the
backend system as possible. Any command sent to the device during operations requires an acknowledgement and
so will reduce the amount of measurements that can be sent back to the backend system. It is therefore a key
ability to fully configure the measurement devices before start of each operation. Before a configurable system is
deployed, it needs to go through a thorough verification process to make sure it is configured (or programmed)
correctly.

Defining the configuration including the data structure for the submitted contents is not a straightforward task. A
number of (possibly conflicting) requirements to the configuration comes from multiple sources: the analysts need
for information, the communication and backend systems, the design of the individual measurement devices, and
from technical issues when combining together several of these devices into a measurement device assembly.
Each operation is different and the configuration must be tailored to the current one.

The effort behind this paper is a result of finding an efficient and consistent approach to achieve quality
configurations for the measurement devices. As is seen from the last part of the paper, more work is needed to
qualify the solution. In particular, the author is very interested in other known uses of this approach.

**PROBLEM**

How can you effectively and consistently validate a complex configuration for a set of measurement devices where
you have multiple requirements for several components of this configuration from various sources?

**FORCES**

Verifying the correctness of a System Configuration is typically done manually by looking at generated log files. It is
a very time consuming process and is prone to human error.

When a Real-Time Measurement system is programmed, the requirements for the configuration often come from
several sources. The system must deliver the data that is requested by clients as well data that is required to
monitor the health and operation of the system itself. These requirements change frequently when the system is
deployed in a different area or used for a different purpose. They also change when some components of the
system are replaced with newer or older versions of those components. Quality requirements are complex and are
only known to experienced specialists.

A typical approach to automating this quality check process is by having a list of error checking functions or
modules (shown in box below). The problem with this approach is that the System Configuration is a complex
object which contains a hierarchy of other complex objects. In our case [Application section], the System
Configuration contains the real-time data formats sent by the various devices (in the system) during different
system states; it also contains configuration details of each of its component devices. Additionally devices have
different firmware versions and there are compatibility issues between firmware versions and the data formats. So
there are complex requirements on any of the component objects within the System Configurations. Therefore
these *Error Checker Functions* tend to be very complicated and difficult to test and maintain.
Another problem of checking the quality requirements as functions is that it forces the business people to go through the software development team every time they want to add a new rule or requirement. The expertise of these rules exists with the people who operate and use the system; not the software developers who write software to configure the system.

**Error Checker Functions**

CheckError1( systemConfiguration, parameters … )
CheckError2(systemConfiguration, parameters … )
...
CheckErrorN(systemConfiguration, parameters … )

**Requirements for System Configuration**

*Figure 2: Diverse requirements are imposed by different users interacting with the system*
1. **APPROACH**

The solution involves identifying the specific objects within the system that have to satisfy the different requirements. The requirements must be expressed as rules in the object-property-constraint pattern (discussed in details in the next section). Users must be able to create rules via a Rule Builder library. The Rule Engine component validates the entire system against the rules (requirements) specified by the users.

### RULE ENGINE APPROACH

1. Identify the objects within the system which have to satisfy requirements.
   a. For each object that has a requirement, identify:
      i. Which property of the object has the requirement and
      ii. What is the constraint imposed by the requirement.

2. Create a Rule Library that enables requirements to be expressed as object-property-constraint rules.

3. Create a Rule Builder library which enables creation of rules for objects within the system.

4. Create a Rule Engine that validates the system against a set of rules (requirements).

The following discussion starts with an overview of the implementation process. The later sections delve into details of the individual steps in the implementation process. Most sections contain examples to clarify details.

2. **IMPLEMENTATION OVERVIEW**

The implementation starts with expressing requirements as rules that the System Configuration must satisfy. A rule is a constraint imposed on a particular property of an object within the System Configuration. The System Configuration itself is the root object which contains all the other objects.

\[ Rule = Object + Property + Constraint \]

#### 2.1 EXPRESSING REQUIREMENTS AS RULES

Let us assume that a client has the requirement that when the Measurement System is stationary, then the Temperature data should be transmitted at least once every 2 seconds. This requirement can be expressed as a rule in the following manner.

**Rule:** The Temperature data point within the Stationary data format should have its Update Rate less than 2 seconds.
Rules can be created for any object at any level of the hierarchy within the System Configuration. As shown in the Figure below, the System Configuration contains Data Formats and other objects. There are multiple Data Formats and each Data Format contains multiple Data Points. The rule above applies to a particular Data Point (Temperature) within a particular Data Format (Stationary Format) inside the System Configuration.

The Rule Library enables creation of such rules for the various Objects in our system.

<table>
<thead>
<tr>
<th>System Configuration</th>
<th>Data Formats</th>
<th>Stationary Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Formats</td>
<td>Data Points</td>
<td>Temperature Blended</td>
</tr>
<tr>
<td>Device List</td>
<td>Slow Moving Data Format</td>
<td>Pressure External</td>
</tr>
<tr>
<td>Device Configurations</td>
<td>Fast Moving Data Format</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3:** A rule can be created for any object inside the hierarchy

### 2.2 VALIDATING REQUIREMENTS VIA RULES

After requirements are entered as rules, the System Configuration object must be validated against these rules. The Rule Engine accomplishes this task. The Rule Engine loads requirements as rules; for each rule it checks whether the System Configuration object satisfies that rule. If a rule is not satisfied, it appears in an error log with a diagnostic message and the details of the error (which object within the hierarchy did not satisfy the rule etc).

### RULE VALIDATION APPROACH

For each rule in list of available rules

If rule is not satisfied by the system configuration

Add rule to the list of errors

Add description of the rule to the error log
3. THE RULE INTERFACE

All rules in the Rule Library implement the Rule Interface. Rules can be applied to various Objects within the System Configuration. The Rule interface for an Object must implement a method called IsSatisfiedBy( Object ). This method validates whether the rule is satisfied by that particular object.

\[
\text{true if the rule is satisfied by the object}
\]

\[
\text{false if the rule is not satisfied by the object}
\]

3.1 A RULE EXAMPLE

Let us assume there is a requirement that when the Measurement System is moving fast, then the size of data packets transmitted by the system should be less than 180 bits (send small data packets because of unreliable wireless communication). We know that when the Measurement System is moving fast, then it transmits data packets according to the data format called the Fast Moving data format. Hence, the requirement stated above could be translated to the following rule:

**Rule:** The Fast Moving data format should have its Size less than 180 bits

\[
\text{Rule: The Fast Moving data format should have its Size less than 180 bits}
\]

This is a rule that validates some property of a Data Format; so it is a Data Format Rule. A Data Format Rule must implement the Rule<Data Format> interface. It has a method IsSatisfiedBy( DataFormat ) which returns whether the rule was satisfied by that particular Data Format.

If we assume that the above mentioned Rule is called the maxSizeRule and the Fast Moving data format is represented by the object fastFormat, then:

\[
\text{maxSizeRule.IsSatisfiedBy (fastFormat)}
\]

Similarly a Data Point Rule (rule that validates a particular Data Point within a Data Format) must implement the IsSatisfiedBy( Data Point ) method.
4. HIERARCHY OF RULE CLASSES

In the previous section, we examined the Rule interface for objects within the System Configuration. We now know that rules can be written for any object; for example:

- If the System Configuration needs to be validated, we write a System Configuration Rule
- If a Data Format needs to be validated, we write a Data Format Rule
- If a Data Point needs to be validated, we write a Data Point Rule

At the heart of the solution lies a relationship between the different types of Rule classes. There is an inheritance hierarchy among these rule classes which is based on how the data is structured in the original system. The data structure of the System Configuration object determines the hierarchy of Rule classes.

4.1 DATA STRUCTURE OF THE SYSTEM CONFIGURATION OBJECT

The figure above shows a part of the data structure of the System Configuration object. The System Configuration contains a list of Data Format objects and a list of Device objects. Each Data Format object contains multiple Data Point objects. The objects have their own properties. The Data Format object contains properties like its Name and Size. The Data Point object has properties like its Name, Size and Update Rate. The Device object has properties like its Name and Firmware Version (the version of the software that is used to operate the device).

Note: In the following discussion, System Configuration is called the Parent Object for Data Format as Data Format objects are contained within System Configuration objects. Data Format is thus the Child Object for System Configuration. Similarly, Data Format is the Parent Object for Data Point and Data Point is the Child Object for Data Format.
4.2 HIERARCHY OF RULE CLASSES

The figure above shows the Rule Class hierarchy. System Configuration Rule is the root. All rules derive from System Configuration Rule; so all rules are System Configuration Rules.

The System Configuration contains a number of Data Formats. So, a Data Format rule is a System Configuration Rule that validates a particular Data Format within the System Configuration.

A Data Point Rule is a Data Format Rule that validates a particular Data Point within the Data Format. Similarly, a Data Point Rule is also a System Configuration Rule because it validates a particular Data Point within a particular Data Format of the System Configuration.

4.3 ANOTHER RULE EXAMPLE

Let us assume a requirement that the System Status Word must be transmitted four times within a data packet when the Measurement System is operating in the Diagnosis Mode. This way the diagnostic tools can figure out if there is something wrong within the system. We know that in Diagnosis Mode, the Diagnostic Data Format is transmitted; hence this requirement can be expressed as the following rule.

\[
\text{Rule } \langle \text{Object} \rangle \text{ is also a Rule } \langle \text{Parent Object} \rangle
\]

A Rule for an Object is also a Rule for its Parent Object because it validates a particular Object within the Parent Object.

A Data Point Rule is a Data Format Rule that validates a particular Data Point within the Data Format. Similarly, a Data Point Rule is also a System Configuration Rule because it validates a particular Data Point within a particular Data Format of the System Configuration.

\[
\text{Rule } \text{System Status Word data point within the Diagnostic data format} + \text{Count} + \text{Equal To 4}
\]

\[
\text{Object Property Constraint}
\]
This is a Data Point Rule, because it validates the Count property of the System Status Word data point. At the same time, it is also a Data Format Rule because it validates the Diagnostic data format. This rule is also a System Configuration Rule because by validating the Diagnostic data format against the requirement, it validates that the System Configuration satisfies the requirement.

The advantage of having a hierarchical rule structure is that while rules can be created for any object within the System Configuration, the Rule Engine can apply any rule to validate the System Configuration object treating it as a System Configuration Rule. This way if there is one Data Point Rule, another Data Format Rule, and a third Device Rule, the Rule Engine simply loads them all as System Configuration Rules and applies the IsSatisfiedBy(SystemConfiguration) method on them.

```
rule.IsSatisfiedBy(systemConfiguration)
```

- true if the rule is satisfied by the system configuration
- false if the rule is not satisfied by the system configuration

It does not matter whether the rule is a Data Point Rule or Data Format Rule or Device Rule.

5. OBJECT IDENTIFIERS

In the previous section, we examined the Hierarchy of Rule objects. We now know that all rules derive from System Configuration Rule, but can be applied to various objects within the System Configuration. An Object Identifier describes which object a rule applies to.

- When creating a Data Format Rule, a Data Format Identifier must be specified which identifies the Data Format that this rule applies to.
- When creating a Data Point Rule:
  - A Data Point Identifier must be specified which identifies the Data Point within a Data Format that this rule applies to.
  - A Data Format Identifier must be specified which identifies the Data Format which contains the Data Point that this rule applies to.

An Object Identifier gets the Object from its Parent Object through the GetFrom(ParentObject) method of the Object Identifier interface. Thus a Data Format Identifier knows how to get that particular Data Format from a System Configuration, and a Data Point Identifier knows how to get the required Data Point from a Data Format.

Object Identifiers enable separation of the rule layer from the application layer. Retrieving different types of objects from the System Configuration for rule validation does not require introduction of a lot of new code into the original objects like System Configuration or Data Format.

```
dataFormatID.GetFrom(systemConfiguration): Returns the particular DataFormat object within the System Configuration object

dataPointID.GetFrom(dataFormat): Returns the particular DataPoint object within the Data Format object
```
6. RULE VALIDATION

6.1 INTERNALS OF THE RULE CLASS

As shown above, a rule for a particular object only needs to find the object within the parent object and determine whether the rule is satisfied for this object.

*Data Format Rule* inherits from the *System Configuration Rule*. As shown below, it has to implement the `IsSatisfiedBy(SystemConfiguration)` method. Inside this method, it extracts the particular *Data Format* that the rule applies to by using the *Data Format Identifier* and then applies the `IsSatisfiedBy(DataFormat)` method.

All Rule objects that derive from *Data Format Rule* only need to implement the `IsSatisfiedBy(DataFormat)` interface. Similarly, the *Data Point Rule* only finds the particular *Data Point* within a *Data Format* and implements a `IsSatisfiedBy(DataPoint)` method.
6.2 A RULE EXAMPLE AGAIN

Let us consider the rule example above (details in Section 4.3). This is a Data Point Rule that applies to the System Status Word data point within the Diagnostic data format. Hence, the Data Point Identifier is the name of the data point, System Status Word, and the Data Format Identifier is the name of the data format, Diagnostic data format.

The figure above shows the sequence of calls that occur during validation of this rule.

- The Rule Engine treats all rules as System Configuration Rule objects. It calls the IsSatisfiedBy(SystemConfiguration) method to find out if this rule is satisfied by the system configuration.
- In the System Configuration Rule class, the IsSatisfiedBy(SystemConfiguration) is a virtual method. Hence, it actually calls the IsSatisfiedBy(SystemConfiguration) method of the Data Format Rule class, which in turn calls the IsSatisfiedBy(DataFormat) method for the Diagnostic data format.
- The IsSatisfiedBy(DataFormat) method of the Data Format Rule class is a virtual method. Hence it calls the IsSatisfiedBy(DataFormat) method of the Data Point Rule class.
- As the rule is actually a Data Point Rule, it executes the IsSatisfiedBy(DataPoint) method for the System Status Word data point. Inside this method it validates whether the count of this data point is equal to 4.
All Rule objects contain a Property and a corresponding Constraint (on that property). The Property class represents a Property of an Object. Examples are Size of a Data Format, Update Rate of a Data Point etc. The Property object also knows how to get the value of a property in an object.

A rule must contain a Constraint object. All Constraint objects implement the method \( \text{constraint.IsSatisfiedBy( propertyValue )} \), where \( \text{propertyValue} \) is the value of the property in the selected object.

There can be different types of constraints depending upon the types of properties. Table 1 shows some examples of Numerical and String Constraints.

When creating a rule, the creator specifies the Property and the Constraint on the property.

### TABLE 1: TYPICAL CONSTRAINTS

<table>
<thead>
<tr>
<th>Numerical Constraint</th>
<th>String Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>LessThan</td>
<td>Contains</td>
</tr>
<tr>
<td>GreaterThan</td>
<td>StartsWith</td>
</tr>
<tr>
<td>EqualTo</td>
<td>EndsWith</td>
</tr>
<tr>
<td>NotEqualTo</td>
<td>EqualTo</td>
</tr>
<tr>
<td>WithinRange</td>
<td></td>
</tr>
</tbody>
</table>

In the rule examples in Sections 2.1, 3.1 and 4.3, we saw the use of the LessThan and EqualTo numerical constraints. Rules related to names and other strings use the string constraints. For example, if there was a rule applicable to all Data Points that contained the word Temperature, then we would use the Contains string constraint.
8. RULE COMPOSITION

8.1 RULE COMPOSITION OPERATIONS

The Rule interface implements the Composite Pattern [1] to enable rules to be combined with other rules. The And, Or and Then operators enable rule composition by creating And Rule, Or Rule, and If Then Rule objects respectively. The Not operator enables creation of negative requirements, that is, conditions that should not be satisfied by the system configuration.

Depending upon the nature of requirements custom rule combination objects can also be created, like the If Then Rule object above. The If Then Rule represents a requirement of the format - If Condition A is satisfied, then Condition B must also be satisfied.

<table>
<thead>
<tr>
<th>Rule Type</th>
<th>How to create</th>
<th>When it is satisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Or Rule</td>
<td>rule1.Or( rule2 )</td>
<td>Satisfied when either rule1 or rule2 is satisfied</td>
</tr>
<tr>
<td>And Rule</td>
<td>rule1.And( rule2 )</td>
<td>Satisfied when both rule1 and rule2 are satisfied</td>
</tr>
<tr>
<td>Not Rule</td>
<td>rule1.Not()</td>
<td>Satisfied when rule1 is not satisfied</td>
</tr>
<tr>
<td>If Then Rule</td>
<td>rule1.Then( rule2 )</td>
<td>Condition is “If rule1 is satisfied, then rule2 must be satisfied”</td>
</tr>
</tbody>
</table>

8.2 RULE COMPOSITION CLASSES

Because of the Composite Pattern (figure above), all the operations on any rule classes result in System Configuration Rule objects. So any type of rule can be combined with any other type of rule. This enables the Rule Builder library to complex rules when required. The figure above (right) also shows an example of the implementation of a composed rule. And Rule must be created with two System Configuration Rules. The
IsSatisfiedBy( SystemConfiguration ) method of AndRule returns true if both rules are satisfied, and false otherwise.

Rules can be combined by any number of operations

\[ \text{rule1.And( rule2 ).Or( rule3 ).And( rule4 )...And( ruleN )} \] is still a System Configuration Rule

8.3 RULE COMPOSITION EXAMPLES

Requirement: When the device is stationary then it must send Average Temperature data.
We know that there are two types of Average Temperature measurements – Average Temperature, and Blended Temperature. The Stationary data format must contain at least one of them.

**Rule 1**

= The Stationary data format + Data Point List + Contains Average Temperature Data Point

**Object** **Property** **Constraint**

**Rule 2**

= The Stationary data format + Data Point List + Contains Blended Temperature Data Point

**Object** **Property** **Constraint**

**Rule** = Rule 1 Or Rule 2

Requirement: The Advanced Computed Pressure measurement of the Smart Pressure Measurement device is valid only for firmware versions 4.0 and above. This is a requirement from the tool manufacturer that some new measurements for the Smart Pressure Measurement device are valid only for the newer firmware versions. So, if the firmware version on the device is an older one, then its data formats should not contain the new measurements.

**IfRule**

= The Smart Pressure Measurement Device + Firmware Version + Less Than 4.0

**Object** **Property** **Constraint**

**Rule 1**

= The Stationary Data Format + Data Point List + Contains Average Computed Pressure Data Point

**Object** **Property** **Constraint**

**ThenRule** = Not( Rule 1 )

**Rule** = IfRule Then ThenRule

Note: The above rule only checks the Stationary data format. To check all data formats via a single rule, they can be combined via the And operator: format1Rule.And( format2Rule ).And( format3Rule )...
RESULTING CONTEXT AND CONSIDERATIONS

The hierarchical rules pattern provides a clean and efficient way of implementing a rule engine to validate complex data structures. It achieves a clear separation between the business logic layer and the applications layer. The hierarchy of rule objects follows the inherent object hierarchy in the data. Rules can be applied to various components of the data structure in an intuitive manner. The design also makes it easy to follow object oriented design principles – low coupling and high cohesion.

The user-defined rules need to be kept simple and small in number. Thousands of complicated rules would be harder to manage and would cause the system to be ineffective; this is a common pitfall for all rule engine based approaches. Like all design patterns, the hierarchical rules pattern is useful only when the data structures and the intentions are suited for it. If the data does not have several levels of objects, then the Specifications pattern [2] might be a better solution. If then intention is not just to validate an object but also to correct the errors (requirements that the object does not satisfy), then the hierarchical rules pattern needs to be combined with other patterns to achieve the intended purpose.

FUTURE WORK

The hierarchical rules pattern lays out the foundation for validating complex objects using user-defined rules. To complete the “user-defined” aspect, the Rule Builder class is still in development; this class which would create rule objects from xml or other user-specified formats. Currently there is a Description Generator class being developed to create automatic custom error messages when rules are not satisfied by an object. Work is also planned on proper user interface design for users to create and edit rules on complex objects.

OTHER POSSIBLE USES

Though our approach was applied for validation of System Configurations for a Real-Time Measurement System, this pattern could possibly be used for validation of any data structure that contains a complex hierarchy of other data structures. The author is very interested in finding our more applications of this pattern. The following are some potential areas where it might be applicable:

1. VALIDATING HOUSE DESIGN PLANS

Our pattern could be used to validate the design plans for a house. A House Design could contain details of the designs for several rooms, which are Room Design plans. Each of the rooms could in turn have design plans for smaller objects like Doors, Windows, and Furniture etc. The requirements for the House Design can come from various people - the customer, the interior decorator, the builder etc. There could be requirements for the entire house (how many stories should it be, how much percentage of the land it should occupy, does it contain a pool, how many bedrooms are there etc.). Also there could be requirements for some very specific objects like the window of the master bedroom (size, position, aspect ratio etc.).

Our pattern can be used to implement requirements as rules at any level of the House Design data structure. All rules are House Design requirements, whether it is a rule imposed on the size of the main window of the master bedroom or on the number of floors of the house. For applying rules on component
objects, there are object identifiers like “master bedroom” which help to identify the particular Room Design object that a rule applies to. There could be a “main window” object within the Room Design which could be used to impose a rule on a smaller component of the design.

2. VALIDATING CAR DESIGN PLANS

A car design plan would also contain the design plans for a complex hierarchy of smaller components, the chassis design, engine design, interior design etc. There would be requirements on many individual components and the requirements would come from various sources – the designers, manufacturers, consumers etc. Our pattern could potentially be used to validate such design plans.

RELATED WORK

This work adapts many ideas from the “Specifications” paper [2] written by Martin Fowler and Eric Evans. In the Specifications paper, the authors write about interfaces like IsSatisfiedBy(Object) and how to combine Specifications via the Composite pattern.

ACKNOWLEDGEMENTS

The author would like to thank colleagues Keith Ray, Andrey Gorodnov and Anjana Nawani at Schlumberger for vital suggestions and inputs throughout the development of the project. The author would like to thank Lise Hvatum, who, through her constant feedback and encouragement helped the author clarify and figure out the various patterns and ideas in his work. She has been a patient and dedicated mentor throughout the development of this paper and this analysis would not have been possible without her.

REFERENCES

1. Design Patterns: Elements of Reusable Object-Oriented Software: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, 1994
3. Analysis Patterns: Reusable Object Models: Martin Fowler, Addison-Wesley, 1997