Rule Engine for Validating Complex Business Objects

DIBYENDUSEKHAR GOSWAMI, goswami.dib@gmail.com

Several systems use complex business objects which need to be validated against multiple requirements before they are deployed. Manually validating these objects can be time-consuming and error-prone. This paper documents an approach that automated this validation process for a complex System Configuration object used to configure a Real-Time Measurement System. The approach enabled validation rules to be separated from business objects and enabled those rules to be specified by business users rather than software developers. The approach uses a Rule Engine pattern to model and apply rules; it takes advantage of the hierarchical nature of the data structure of the business object being validated.

Categories and Subject Descriptors: **D.2.11 [Software Engineering]**: Software Architectures—*Patterns*; **K.6.3 [Management of Computing and Information Systems]**: Software Management—*Software Development*; **D.2.4 [Software Engineering]**: Software/Program Verification—*Validation*

General Terms: Design, Documentation

Additional Key Words and Phrases: Rule engine, Business rules, Rule-based validation

1. INTRODUCTION

Consider a scenario where Alice, an oilfield engineer, is preparing a *Real-Time Measurement System* for deployment. This system consists of various measurement devices that acquire, transmit and record data. The system will be deployed in a remote drilling operation where it has very limited communication ability with the control center and must operate autonomously. *Therefore, it is of the utmost importance that the system is configured correctly before it is deployed— that is, the System Configuration file is thoroughly verified against all applicable requirements*.

Alice attempts to validate this *System Configuration* file, an XML file, by manually loading it up in a viewer and soon realizes that it is a very time-consuming and error-prone task. The *System Configuration* must satisfy several diverse and possibly conflicting requirements that come from different sources (Figure 1). The System must be configured to serve the needs of the Data Analysts, Operation Supervisors and Acquisition Supervisors; at the same time it needs to follow all the device specifications by the manufacturers of the different devices. Some devices are not compatible with each other when they are not running particular firmware versions. Moreover, the validation requirements change depending upon the location where the Measurement System is being deployed (Figure 2). Alice needs *software that can help automate and manage this task of validating the complex System Configuration file*.

The above scenario is an example where:

- *A business object (the System Configuration) is very complex and contains a hierarchy of configurable components (Measurement Devices).*
- *Validating the state of this business object before deployment is of utmost importance to the system.*
- *The business requirements often validate one or more components within the business object.*
- *The requirements for one component can be affected by the state of another component.*
- *Requirements come from different sources and change rapidly with various factors like time, location, or purpose for which the system is being used.*

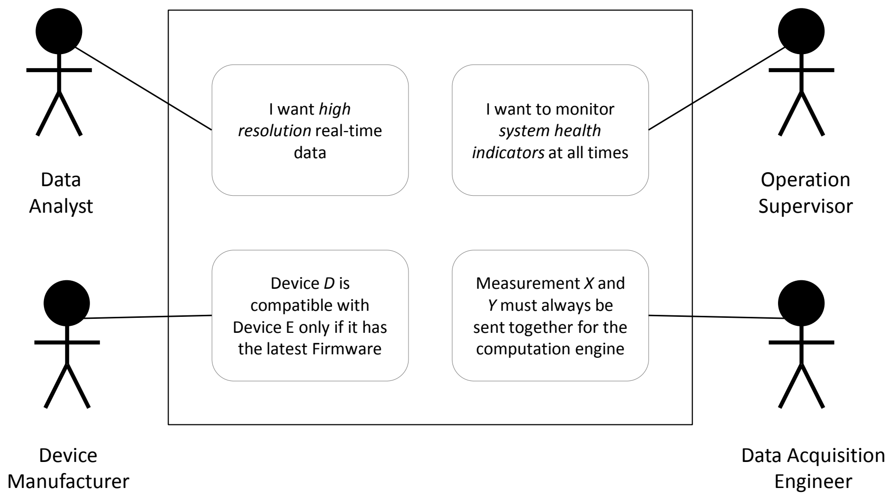

I want *high resolution* real-time data
I want to monitor *system health indicators* at all times
Device *D* is compatible with Device E only if it has the latest Firmware
Measurement *X* and *Y* must always be sent together for the computation engine
Data Analyst
Operation Supervisor
Device Manufacturer
Data Acquisition Engineer

Figure 1: Business requirements are imposed by various sources

A typical approach to automating the validation process is by implementing the business rules within the body of methods of the business objects. In this approach, however, the business rules are tied to the objects which they apply to. *Hence it is hard to maintain code when business rules change very often*, because changing one rule or condition can cause side-effects and compromise code quality. Moreover, this approach would lead to *Scattered Rules* (Arsanjani 2001), where business rules are typically scattered across the application and are difficult to manage.

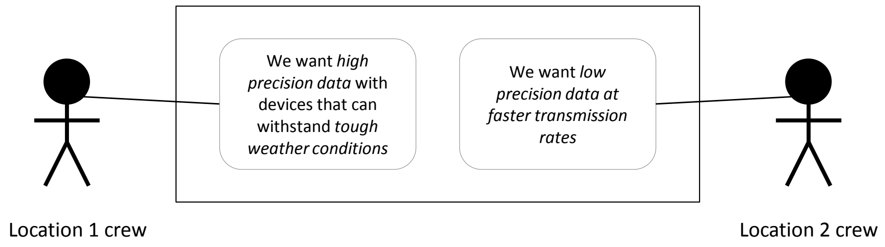
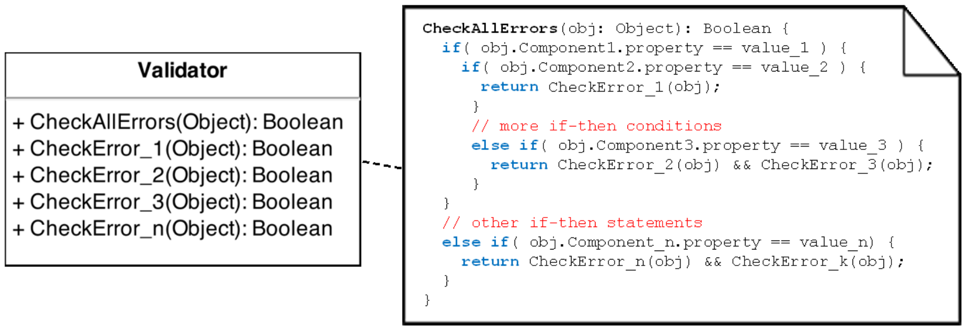

We want *high precision data* with devices that can withstand *tough weather conditions*
We want *low precision data at faster transmission rates*
Location 1 crew
Location 2 crew

Figure 2: Business requirements change with location of operation

Another approach for validating business objects is by implementing a *Validator object* with various methods to check and validate the business object (Figure 3). This approach is similar to the RULE


Validator
+ CheckAllErrors(Object): Boolean
+ CheckError_1(Object): Boolean
+ CheckError_2(Object): Boolean
+ CheckError_3(Object): Boolean
+ CheckError_n(Object): Boolean

```
CheckAllErrors(obj: Object): Boolean {
  if( obj.Component1.property == value_1 ) {
    if( obj.Component2.property == value_2 ) {
     return CheckError_1(obj);
    }
    // more if-then conditions
    else if( obj.Component3.property == value_3 ) {
      return CheckError_2(obj) && CheckError_3(obj);
    }
  }
  // other if-then statements
  else if( obj.Component_n.property == value_n) {
    return CheckError_n(obj) && CheckError_k(obj);
  }
}
```

Figure 3: Code cluttered with If-Then statements with single Validator object

METHOD PATTERN (Arsanjani 2001). This approach is better than having rules embedded in the business object itself as the rules are not strongly coupled with the business object and can be changed without side-effects to the object. However, in our case, the validation rules for the business object change depending upon multiple factors including the interaction between the different component objects within the business object. *Hence, this approach would cause the code to be cluttered with nested If-Then statements attempting to accommodate the various scenarios under which a rule is applicable*. Moreover, business users cannot change the rules themselves. They would still have to submit requests to the software development team whenever they need to add, delete or update any validation rules.

In this paper, we document an approach that we used to successfully automate this validation process for a complex business object— a System Configuration object— used to configure a Real-Time Measurement System.

In our approach, we use patterns in the *Rule Object Pattern Language* (Arsanjani 2001). We use the SPECIFICATION PATTERN (Fowler and Evans) to check whether our business object satisfies a particular rule or specification. We use the RULE OBJECT pattern to create a *rule object* corresponding to every validation requirement, thus separating the validation rules from the objects that they validate. We use the COMPOSITE RULE OBJECT pattern or the COMPOSITE SPECIFICATIONS pattern to enable definition of complex rules by composing simple rules. We use the RULE ENGINE pattern to externalize rules into files and load them during run-time. Together these patterns provide a robust and extensible architecture for managing validation rules in a business context which keeps changing rapidly with time and location. We created a *Domain Specific Language* which allowed users to specify rules in (*Object, Property, Constraint*) sets. We also took advantage of the hierarchical nature of the business object to create a *Hierarchical Rules* structure.

The remainder of this paper is organized as follows.
- *The Problem (Section 2) and Forces (Section 3) sections define the problem and describe the various factors and aspects that need to be resolved.*
- *The Solution (Section 4) describes our approach, starting with an overview and subsequently explaining the details in several sub-sections.*
- *The Resulting Context (Section 5) describes how the various forces in the problem have been addressed by our solution.*
- *The Limitations and Future Work (Section 6) explores on the limitations of this solution and what could be done in the future to improve it.*

2. PROBLEM

When the requirements for validating a complex business object come from multiple sources and change rapidly, then the validation process can easily become unmanageable and messy. How can you automate such a validation process so that it is robust, extensible and easy to manage?

3. FORCES

Rules can be applied at various levels of the hierarchy of the business object. Some rules validate the whole business object whereas other rules validate various components of the object. As shown in Figure 4, the *System Configuration* object contains *Data Format* objects, which in turn contain *Data Point* objects. Some rules might validate individual *Data Points*, whereas other might validate multiple *Data Formats*. Yet other rules might validate the whole *System Configuration* itself.

Rules originate from various sources. As shown in Figure 2, validation rules can come from different sources— Data Analysts, Data Acquisition Engineers, Device Manufacturers, Operation Supervisors etc.

Rules need to change rapidly depending upon several factors. Depending upon the location, time or purpose of operation of the *Measurement System*, the validation rules can change. So the user should be able to choose which rules should be applied when. Moreover, rules change when newer versions of device hardware and software are available.

Rules can be pretty complicated often depending upon complex interactions between the various components in the system. Validation rules on certain components of the business object can depend upon the state of other components within the business object. For example, if the firmware (software) version of a device *Device-A* is less than 5.0 then the *Advanced Temperature Data Point* cannot be used for another device *Device-B*.

4. SOLUTION

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* language (discussed in details in the next section). Users are able to create rules via a *Rule Builder* library. The *Rule Engine* component validates the overall data structure against the rules (requirements) specified by the users.

Rule Engine Approach

identify the *objects* within the system which have to satisfy requirements.

for each *object* that has a requirement, **identify**:
    which *property* of the object has the requirement, and
    what is the *constraint* imposed by the requirement.

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

create a *Rule Builder Library* which enables creation of rules for objects within the system.

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.

4.1 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*. This object can be any component within the business object being validated. The System Configuration itself is the root object which contains all the other objects.

Rule = Object + Property + Constraint

4.1.1 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

Rule	**=**	**The Temperature data point within the Stationary data format**	**+**	**Update Rate**	**+**	**Less Than 2 sec**
		Object		*Property*		*Constraint*

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.

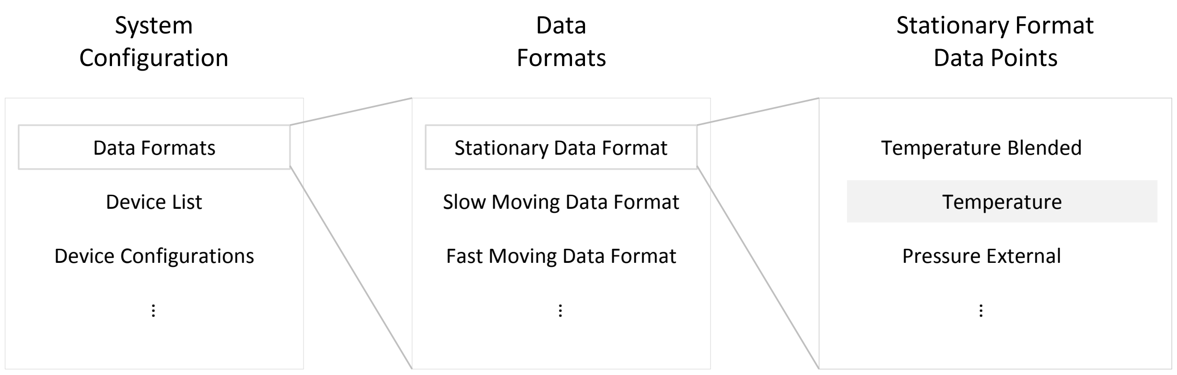


Figure 4: A rule can be created for any object inside the hierarchy

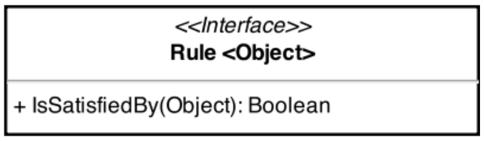
4.1.2 Validating Requirements via Rules

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 component within the hierarchy did not satisfy the rule etc).

Rule Validation Approach

```
for each rule in list of available rules, do:
    check if rule is satisfied by the system configuration.
    if rule is not satisfied:
        add rule to the list of errors.
        add description of the rule to the error log.
```

4.2 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.

$$\text{rule.IsSatisfiedBy (object)} \begin{cases} \textit{true} \text{ if the rule is satisfied by the object} \\ \textit{false} \text{ if the rule is not satisfied by the object} \end{cases}$$

4.2.1 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

$$\textbf{\textit{Rule}} = \underset{\textit{Object}}{\textbf{The Fast Moving data format}} + \underset{\textit{Property}}{\textbf{Size}} + \underset{\textit{Constraint}}{\textbf{Less Than 180 bits}}$$

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*(*Data Format*) 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:

$$\text{maxSizeRule.IsSatisfiedBy (fastFormat)} \begin{cases} \textit{true} \text{ if the size of fastFormat is less than 180 bits} \\ \textit{false} \text{ if the size of fastFormat is not less than 180 bits} \end{cases}$$

Similarly a *Data Point Rule* (rule that validates a particular *Data Point* within a *Data Format*) must implement the *IsSatisfiedBy*(*Data Point*) method.

4.3 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 created for any object. For example:

– *If the System Configuration needs to be validated, a System Configuration Rule should be created.*
– *If a Data Format needs to be validated, a Data Format Rule should be created.*
– *If a Data Point needs to be validated, a Data Point Rule should be created.*

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 business object being validated by the rules. The data structure of the *System Configuration* object determines the hierarchy of Rule classes.

4.3.1 Data Structure of the System Configuration object

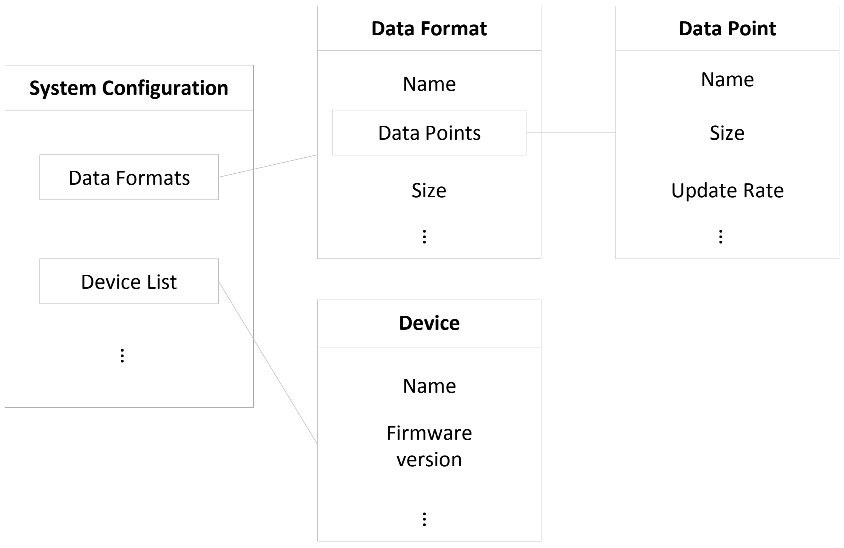


Figure 5: Data Structure of the System Configuration object

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*.

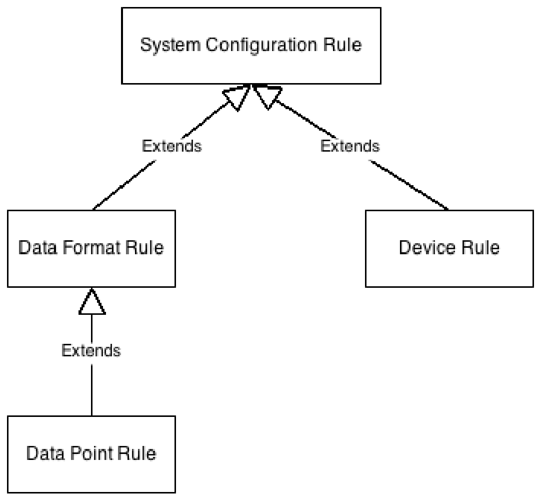


Figure 6: Hierarchy of Rule Classes

4.3.2 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*.

Rule <Object> *is* also a Rule <Parent Object>

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*.

4.3.3 A 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 helps the diagnostic tools detect if there are any components malfunctioning 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.

Rule	**=**	**The System Status Word data point within the Diagnostic data format**	**+**	**Count**	**+**	**Equal To 4**
		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(System Configuration) method on them.

$$\text{rule.IsSatisfiedBy (system configuration)} \begin{cases} \textit{true} \text{ if the rule is satisfied by the system configuration} \\ \textit{false} \text{ if the rule is not satisfied by the system configuration} \end{cases}$$

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

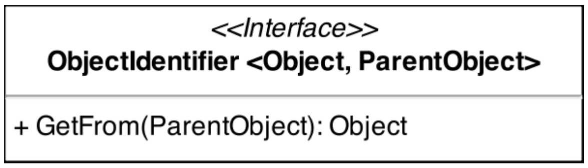
4.4 Object Identifiers

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.

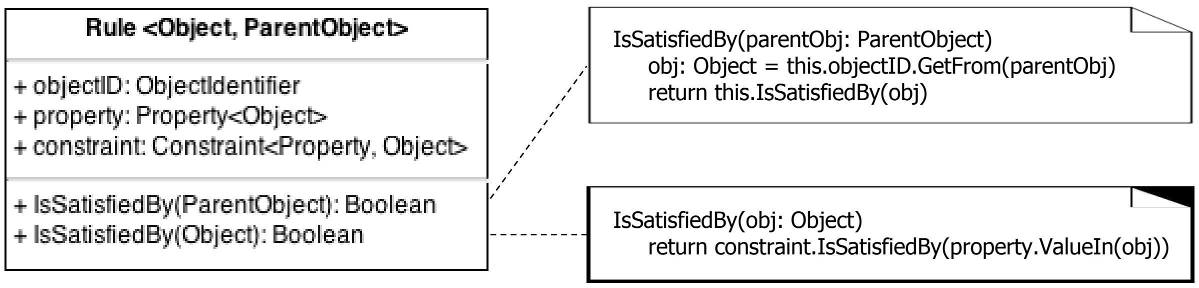
When creating a *Data Point Rule*:
– A *Data Point Identifier* must be specified which identifies the *Data Point* within a *Data Format* that this rule applies to.
– A *Data Format Identifier* must be specified which identifies the *Data Format* which contains the *Data Point* that this rule applies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


<<Interface>>
ObjectIdentifier <Object, ParentObject>
+ GetFrom(ParentObject): Object

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


Rule <Object, ParentObject>
+ objectID: ObjectIdentifier
+ property: Property<Object>
+ constraint: Constraint<Property, Object>
+ IsSatisfiedBy(ParentObject): Boolean
+ IsSatisfiedBy(Object): Boolean

```
IsSatisfiedBy(parentObj: ParentObject)
    obj: Object = this.objectID.GetFrom(parentObj)
    return this.IsSatisfiedBy(obj)
```

```
IsSatisfiedBy(obj: Object)
    return constraint.IsSatisfiedBy(property.ValueIn(obj))
```

Figure 7: Details of Rule class

4.5 Rule Validation

4.5.1 Details of the Rule Class

As shown in the figure above, a rule for a particular object first needs to find the object within the parent object and then determine whether the constraint is satisfied by its property.

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 directly 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.

4.5.2 A Rule Example

Rule	**=**	**The System Status Word data point within the Diagnostic data format**	**+**	**Count**	**+**	**Equal To 4**
		Object		*Property*		*Constraint*

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.

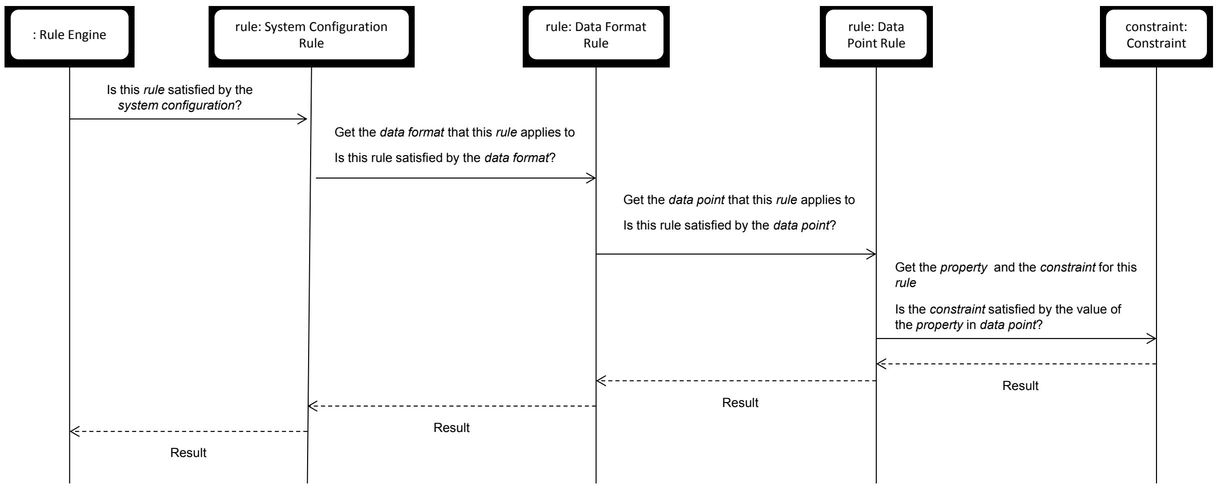

: Rule Engine
rule: System Configuration Rule
rule: Data Format Rule
rule: Data Point Rule
constraint: Constraint
Is this *rule* satisfied by the *system configuration*?
Get the *data format* that this *rule* applies to
Is this rule satisfied by the *data format*?
Get the *data point* that this *rule* applies to
Is this rule satisfied by the *data point*?
Get the *property* and the *constraint* for this *rule*
Is the *constraint* satisfied by the value of the *property* in *data point*?
Result
Result
Result
Result

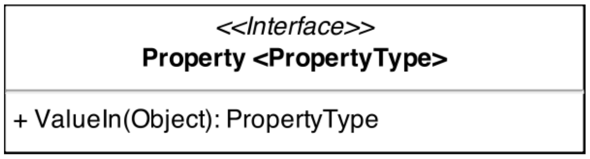
Figure 8: Sequence of calls for validating a Data Point Rule

The Figure 8 shows the sequence of calls that occur during validation of the *Data Point* rule example:

– The *Rule Engine* treats all rules as *SystemConfigurationRule* objects. It calls the `IsSatisfiedBy(SystemConfiguration)` method to find out if this rule is satisfied by the system configuration.
– In the *SystemConfigurationRule* class, the `IsSatisfiedBy(SystemConfiguration)` actually calls the `IsSatisfiedBy(SystemConfiguration)` method of the *DataFormatRule* class, which in turn calls the `IsSatisfiedBy(DataFormat)` method for the *Diagnostic Data Format*.
– The `IsSatisfiedBy( DataFormat )` method of the *DataFormatRule* class calls the `IsSatisfiedBy(DataFormat)` method of the *DataPointRule* class.
– As the rule is actually a *DataPointRule*, 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*.

4.6 Properties and Constraints

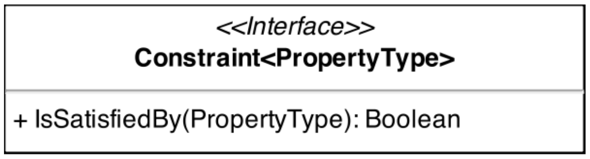
All Rule objects contain a *Property* and a corresponding *Constraint* on that property. The *Property* class represents a *Property* of an *Object*. Examples are Size of a Data Format, Update Rate of a Data Point etc. The *Property* object also knows the value of the property in the selected object.



property.ValueIn(object): Returns the value of a property in a particular object

The Rule object also contains a Constraint object. All Constraint objects must implement the function `constraint.IsSatisfiedBy(property.value)`.

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.

constraint.IsSatisfiedBy(property.ValueIn(object)): Returns whether the constraint has been satisfied by the value of a particular property in the object

Table 1: Typical Constraints

NUMERICAL CONSTRAINT	STRING CONSTRAINTS
LessThan	Contains
GreaterThan	StartsWith
EqualTo	EndsWith
NotEqualTo	EqualTo
WithinRange	

In the rule examples in Sections 4.1.1, and 4.2.1, the rules were created with a *LessThan* constraint on the *UpdateRate* and *Size* properties. In the rule example in Section 4.3.3, the *EqualTo* constraint was used on a *Count* property. 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 the *Contains* string constraint can be used.

The Constraints and Properties depend upon the domain of application and are an important part of the Domain Specific Language.

4.7 Rule composition

4.7.1 Rule Composition Operators

The Rule interface follows the COMPOSITE PATTERN (Eric Gamma et. al. 1994) which enables combination of rules with other rules. The composite rules also implement the *Rule* interface. Hence they behave just like any other rules. This is similar to the COMPOSITE SPECIFICATIONS pattern described in Specifications (Fowler and Evans) or the COMPOSITE RULE OBJECT pattern described in the *Rule Object Pattern Language* (Arsanjani 2001).

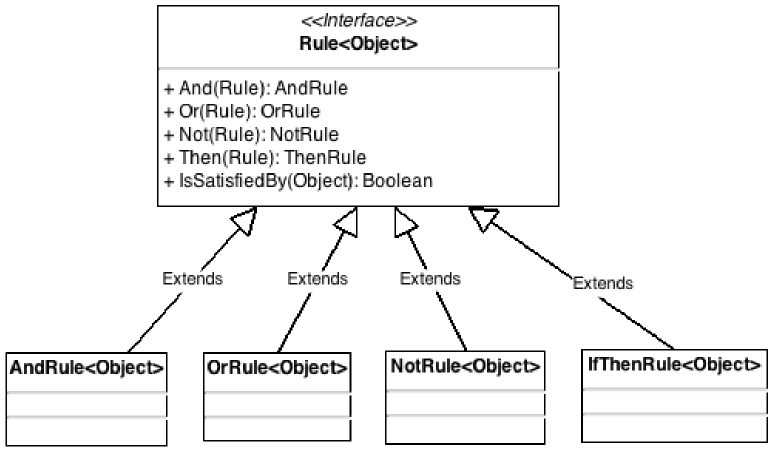
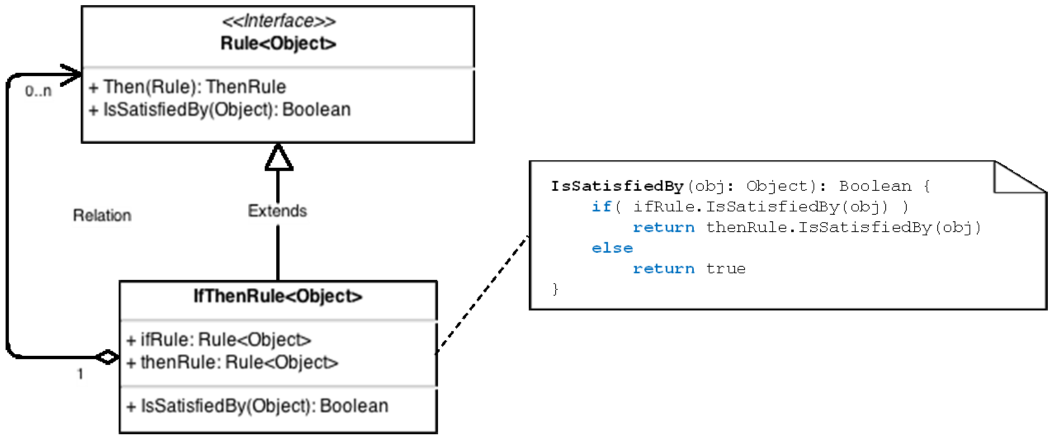


Figure 9: Rule Composition via the Composite Pattern

The *And*, *Or*, *Not* and *Then* methods enable rule composition by creating *AndRule*, *OrRule*, *NotRule* and *IfThenRule* objects respectively. The figure below shows how the *IfThenRule* class implements the *Rule* interface.



```
IsSatisfiedBy(obj: Object): Boolean {
    if( ifRule.IsSatisfiedBy(obj) )
        return thenRule.IsSatisfiedBy(obj)
    else
        return true
}
```

Figure 10: Example of IfThenRule implementation

The Table below shows details of the different composite rule types.

Rule Type	How to create	Description
Or Rule	rule1.Or(rule2)	Satisfied when either rule1 or rule2 are satisfied
And Rule	rule1.And(rule2)	Satisfied when both rule1 and rule2 are satisfied
Not Rule	rule1.Not()	Satisfied when rule1 is not satisfied
If Then Rule	rule1.Then(rule2)	Condition: If rule1 is satisfied, then rule2 must be satisfied

The various operators like *Or*, *And*, *Not* and *IfThen* augment the *Domain Specific Language* as they enable creation of complex rules by combining simple using these operators.

Rule combination operators can be chained any number of times

rule1.And(rule2).Or(rule3).Not(rule4)…And(ruleN) is still a *SystemConfigurationRule.*

4.7.2 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 Device Manufacturer because 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 Advanced 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 )…`

5. RESULTING CONTEXT

Our approach resolved the various forces of our validation problem.

The use of Rule objects creates a clean separation between the business objects and the rules that validate those objects. The Hierarchical Rules structure (Section 4.3) enables creation of rules at any level of the hierarchy within the System Configuration object. Since all rules derive from *SystemConfigurationRule*, hence the Rule Engine does not need to treat them differently. All rules validate some parts or the whole of the *System Configuration* object.

The *Object-Property-Constraint language makes it easy to represent as well as create rules.* Users can create rules via a simple User Interface that enables them to choose the *Objects*, their corresponding *Properties* and what *Constraints* they would apply to those *Properties*. The RuleBuilder library handles the job of exporting to and importing rules from XML files. Thus the different sources for the validation requirements can create the rules that are relevant to them. The Device Manufacturer can create the rules relevant to the devices. The *Data Analysts*, or *Operation Supervisors* can create rules relevant to their operations. The Software Development team does not need to create and maintain rules. Their responsibility is limited to maintaining the *Domain Specific Language* that is used to create rules.

The use of a Rule Engine helps externalize rules and load them during run-time. Thus, different users can specify their own rule files that help them validate the business objects according to their customized requirements.

The use of Rule Composition enables creation of complex rules by composing simple rules via the And, Or, Not and IfThen operators. These operators are a part of the *Domain Specific Language* and can vary depending upon the domain of application. Thus, all complex interactions between various components can be easily managed by isolating the conditions for each component and combining them via the operators.

6. LIMITATIONS AND FUTURE WORK

As with most Rule-Engine approaches, for our approach, the user-defined rules would need to be kept simple and small in number (Fowler 2009). Thousands of complicated rules would be harder to manage and would cause the system to be ineffective. There are several best practices established around creation and management of business rules (Gladys 2003, Ronald 2003, Wan-Kadir and Loucopoulos 2004, 2008). These must be followed in order to ensure the success of our approach. Currently our approach is only used to perform validation on the business object but we do not address the issue of what should be done if the state of the object is not valid. Currently our rule objects have a *ErrorDescription* property which generates an error message whenever the validation rule fails. The approach would be very useful if it could automatically correct the state of the business object based on the rules that failed. Constraint-based optimization approaches can be explored.

7. ACKNOWLDEGEMENTS

The author would like to thank colleagues Keith Ray, Andrey Gorodnov and Ed Tollefsen 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 describe the 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. The author is also grateful for the invaluable comments and suggestions from Filipe Correia, Joshua Kerievsky, Juan Reza, Jeffrey Overbey and Russ Rubis at the author's workshop session at PLOP 2013.

REFERENCES

Arsanjani, A. 1998. GOOD: Grammar-oriented Object design, Position Paper for *OOPSLA Workshop on Metadata and Active Object Models*, 1998, Vancouver, British Columbia.
Arsanjani, A. 2001. Rule object 2001: A pattern language for adaptive and scalable business rule construction, In *Proceedings of the 8th Conference on Pattern Languages of Programs (PLOP 2001)*. IEEE Computer Society, 370 - 402.
Fowler, M. 1997. Analysis Patterns: Reusable Object Models. Addison-Wesley.
Fowler, M. 2003. Patterns of Enterprise Application Architecture, Addison-Wesley Professional.
Fowler, M. 2009. Should I use a Rule Engine. Retrieved May 01, 2014 from MartinFowler.com: http://martinfowler.com/bliki/RulesEngine.html
Fowler, M., Parsons, R. 2011. Domain Specific Languages. Addison-Wesley.
Fowler, M and Evans, E. Specifications. Retrieved May 01, 2014 from Martin Fowler: Articles: http://www.martinfowler.com/apsupp/spec.pdf
Gladys S. W. Lam. 2003. The Hidden Secrets about a Business Rule, *Business Rules Journal*, Vol. 4, No. 7 (July 2003)
Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1994. Design Patterns: Elements of reusable Object-oriented Software. Addison-Wesley.
Loucopoulos, P., Kadir, W. M. 2008. BROOD: Business Rules-driven Object Oriented Design. Journal of Database Management (JDM), 19(1), 41-73.
W. M. N. Wan-Kadir and Pericles Loucopoulos. 2004. Relating evolving business rules to software design. J. Syst. Archit. 50, 7 (July 2004), 367-382. DOI=10.1016/j.sysarc.2003.09.006 http://dx.doi.org/10.1016/j.sysarc.2003.09.006
Ronald G. Ross. 2003. *Principles of the Business Rule Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA
Yoder, J.W., Balaguer, F., Johnson, R. 2001. Adaptive Object Models for Implementing Business Rules. Position Paper for *Third Workshop on Best-Practices for Business Rules Design and Implementation, OOPSLA (2001)*

Appendix

1. Definition of Terms

Real-Time Measurement System	A system or suite that consists of various measurement devices that acquire, transmit and record data. Most of the systems referred to in this paper are deployed in environments where the measurement system 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 systems used in wireless submarine ROV operations, space operations with communication between an earth station and remote equipment, and measurement and logging while drilling operations.
Device	This represents a physical measurement device. A device typically has multiple sensors which can measure physical properties like temperature, pressure, resistivity etc. The software or firmware that a device runs can be updated to provide enhanced functionalities and advanced measurements. Hence, the firmware version of a device is very important entity
Data Format	This represents the structure of a data packet that is transmitted by the Real-Time Measurement System to the backend control and recording system. The Measurement System has different data formats for the different types of data depending upon the purpose and location of its deployment. The resolution of the data captured also varies depending upon the movement of the Measurement System. So, if the System is moving fast, it automatically starts sending 'Fast Moving Data Format' data packets. Or, if the System is in Diagnosis mode, it starts sending 'Diagnosis Data Format' data packets.
Data Point	This represents a single data field within a Data Format. A data point can represent several entities. Typically, a data point is a physical measurement (like Temperature, Pressure etc.) or System Health Monitoring data (like Status words from various devices). A Data Point has several properties like Name, Identifier, Size, Device List (which devices are capable of acquiring this data) etc.

2. Validating House Design Plans - A Similar Problem

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 solution documented in this paper could be used to validate the design plans for a house. A *House Design* could contain details of the designs for several rooms. Each of these rooms would have *Room Design* plans. 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 and so on. There could be requirements for the entire house (the number of stories it should have, the percentage of land it should occupy, options and specifications for a pool, the number of bedrooms it should have, etc..). Also there could be requirements for some very specific objects like the window of the master bedroom (size, position, aspect ratio and so on).

Using our approach requirements can be implemented 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.