Rule Engine Design

Michael Walfall

Version 0.5.0

Table of Contents

Requirements 2

Component and Class Diagrams 2

Business Layer Class Diagram 3

Rules Layer Class Diagram 4

Rules Engine (Service) Layer 5

Rules and RuleSets Class Diagram (Detailed) 6

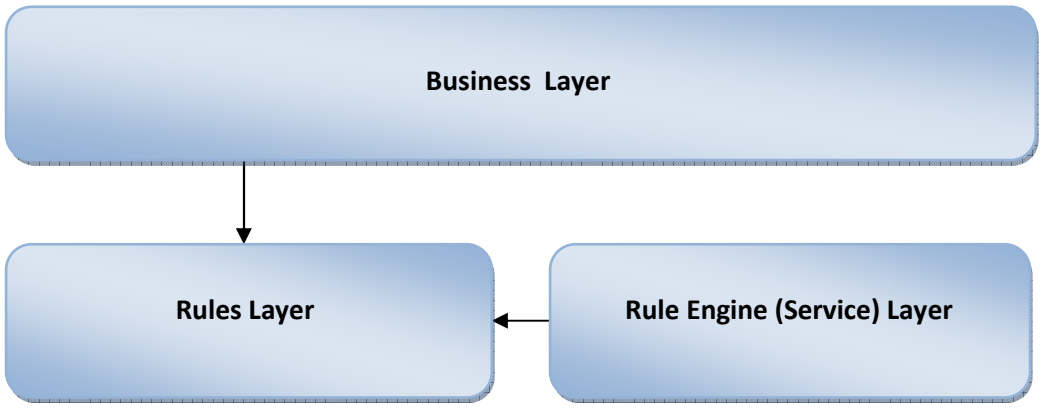
Proof of Concept: And Operation Between Rules 7

Complete Rules Engine Class Diagram 10

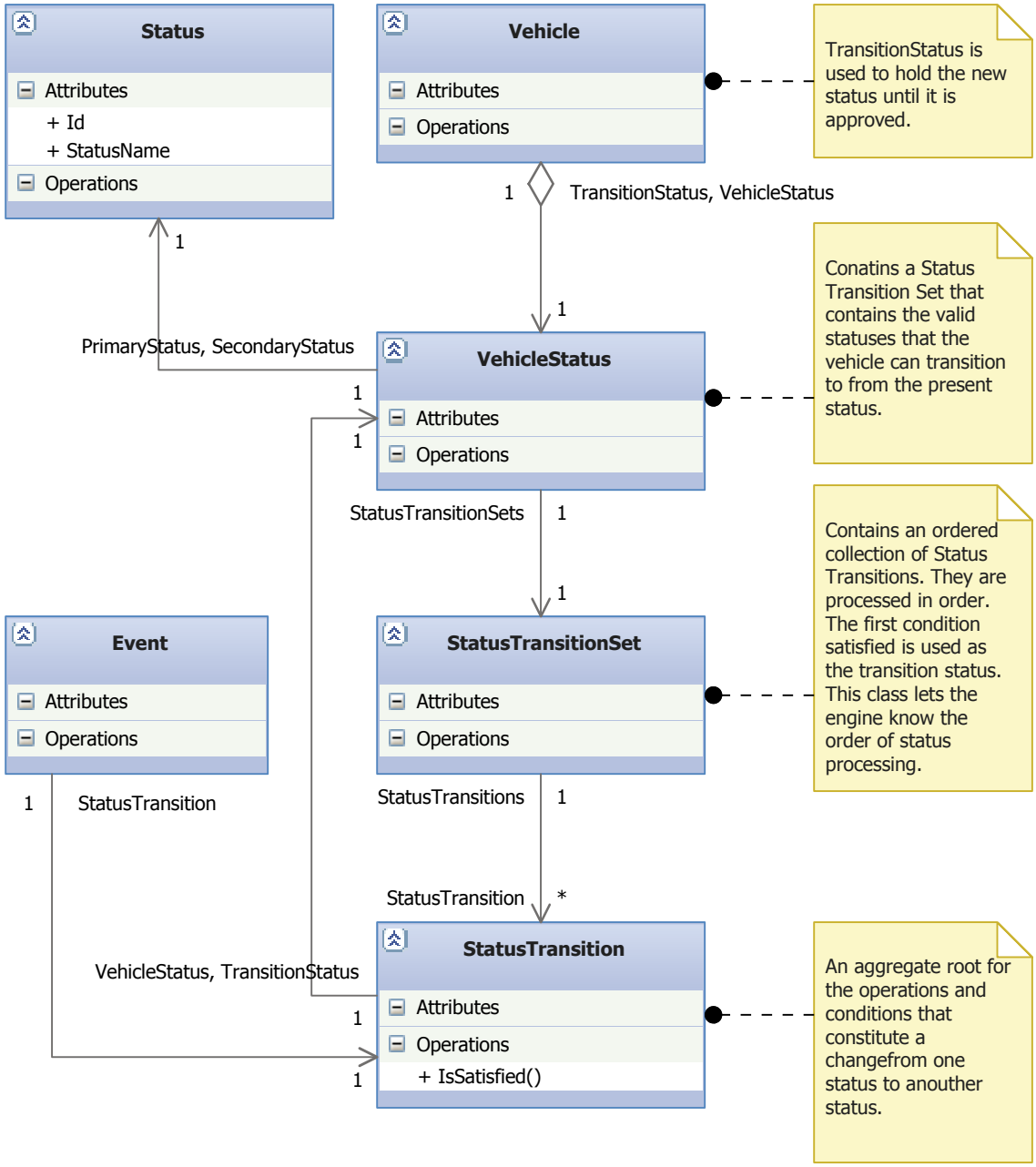
Requirements

- Given a present status, the application must be able to determine the new status of a vehicle when information is submitted for processing.
- A user can use an Event to change the status of a vehicle. When an Event is submitted, the application must be able to determine if a valid status transition is being requested.
- It must be possible to add a status and modify a status without writing code.
- It must be possible to add a rule and modify a rule without writing code.
- It should be possible to perform validation against different entity types without writing code to modify the rule engine.

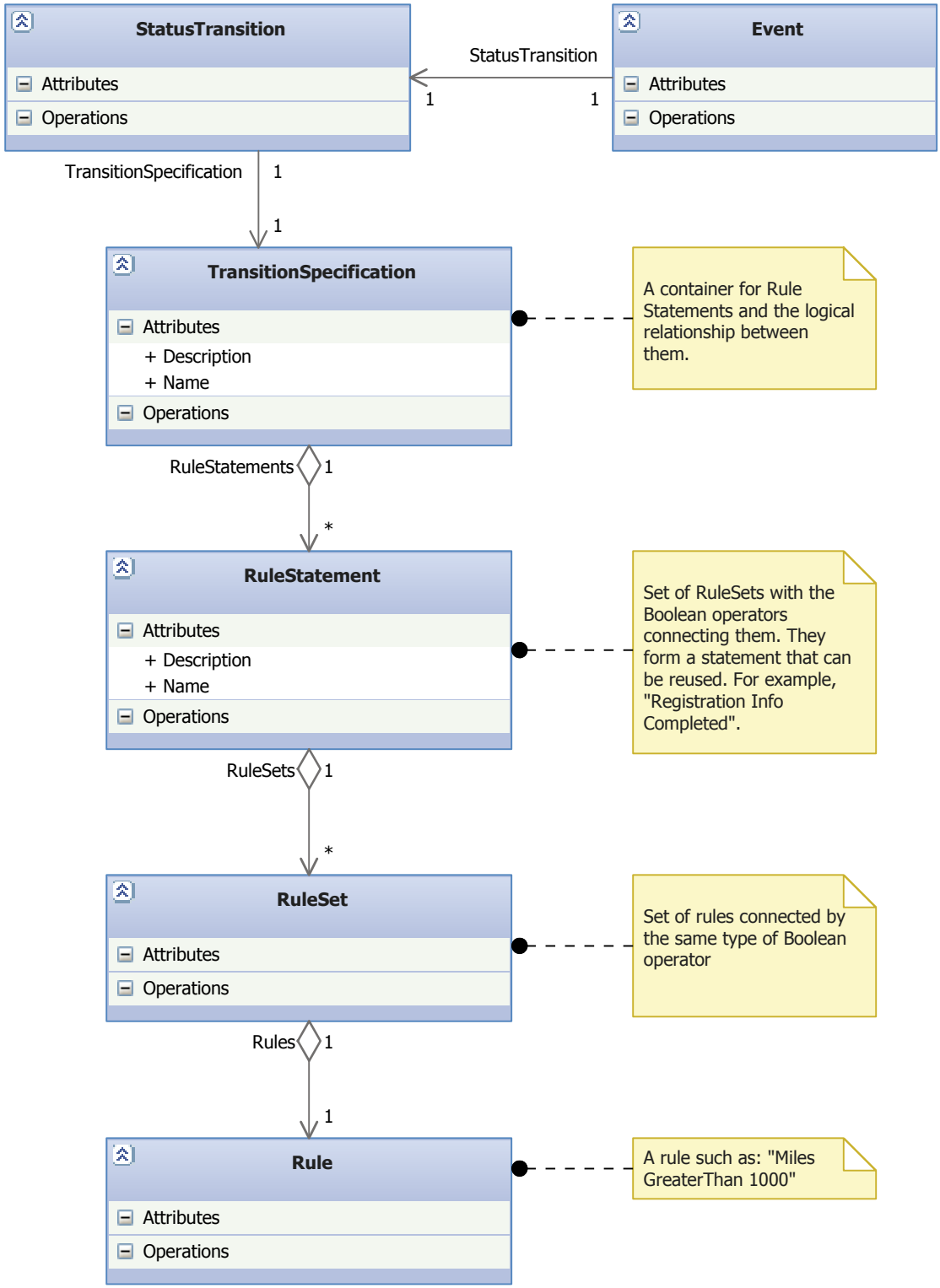
Component and Class Diagrams



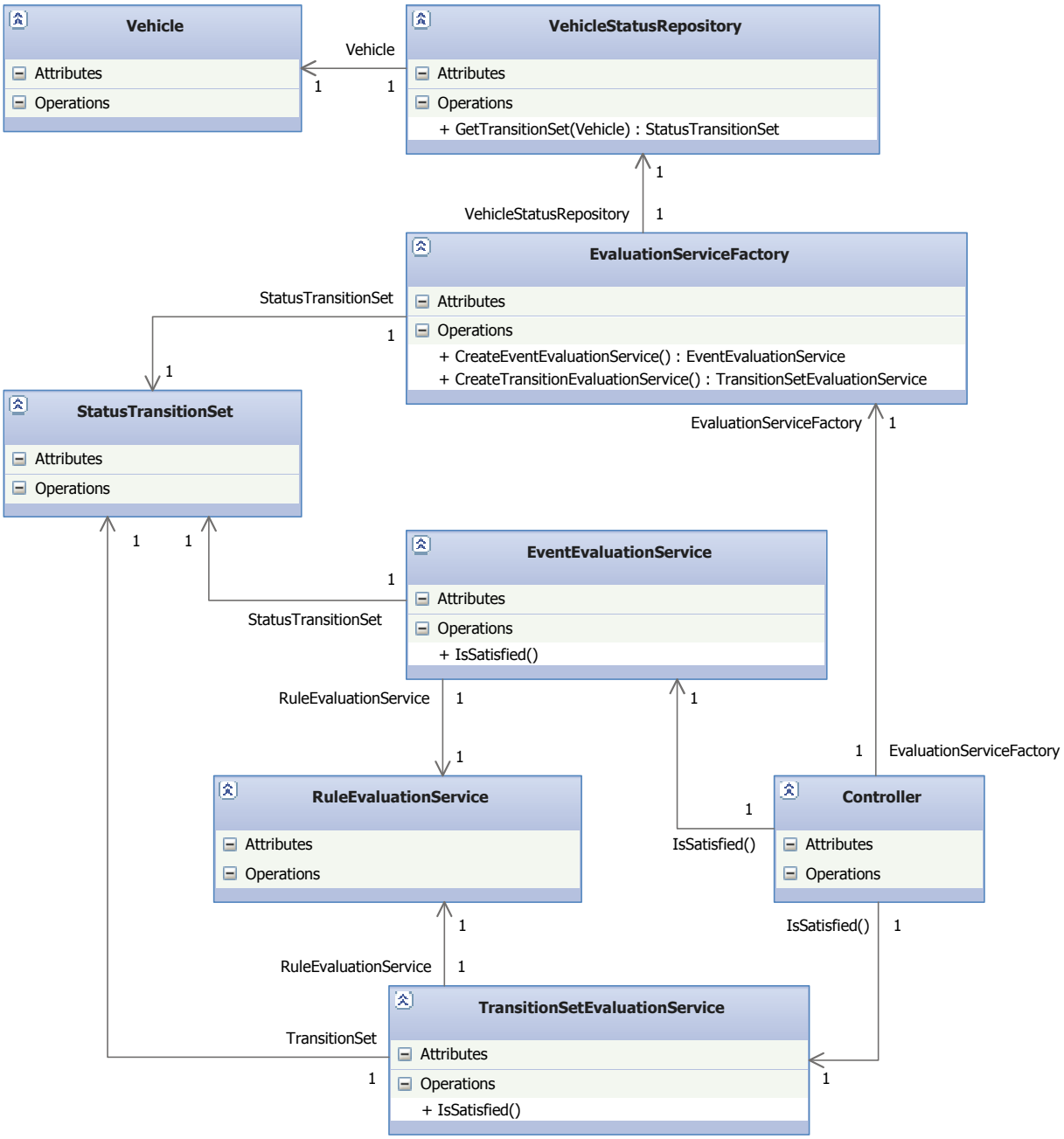
Business Layer Class Diagram



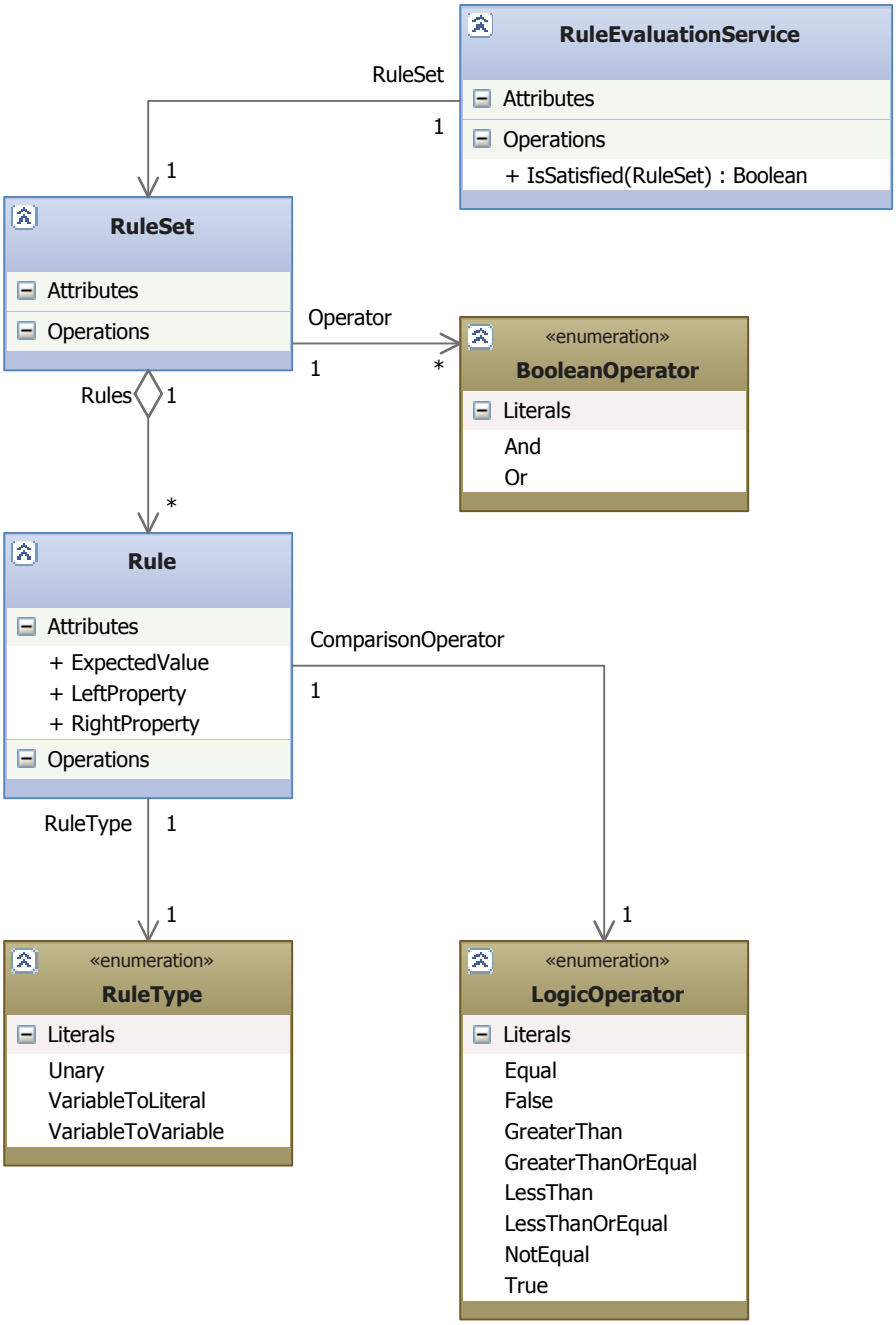
Rules Layer Class Diagram



Rules Engine (Service) Layer



Rules and RuleSets Class Diagram (Detailed)



Proof of Concept: And Operation Between Rules

In this section shows a proof of concept implementation of the AND operation between three rules.

Listed below are examples of rules. They are built using the .NET ExpressionType enumeration class. Each rule element will have a rule type that will inform the engine of how the element is to be processed.

Rule Examples	
Miles *GreaterThan* **1000**	RuleType.VariableToLiteral
ServiceDate *Equal* **ActivatedDate**	RuleType.VariableToVariable
ServiceDate *True*	RuleType.Unary

Rules will be housed within a rule set in accordance with the logical operation between them. The table below shows how the rules would be broken up into rule sets.

(A & B & C) | D	A,B,C in one RuleSet; D in one RuleSet.
(A & B) | (C & D)	A, B in one RuleSet; C, D in one RuleSet.

This sort of setup enables the application to take advantage of Linq Expressions to process the information. The Linq method **`All()`** can be used as an **AND** operator and the Linq method **`Any()`** can be used as an **OR** operator.

The following unit test serves as a proof of concept of the All() method being used as an AND operator between rules. The rule set being processed is as follows:

PlateNumber == “123-abc” AND Transmission != “2345678” AND Miles > 2000

The three rule objects are shown below. The vehicle and the rule set are passed as parameters to the `RuleValidationService.IsSatisfied()` method in the latter part of the code.

```
[TestMethod()]
public void IsSatisfiedTest()
{
    Vehicle vehicle = new Vehicle
    {
        PlateNumber = "123-abc",
        Transmission = "2343678",
        Miles = 3000
    };

    List<Rule> ruleList = new List<Rule>{
        new Rule
        {   RuleType = RuleType.VariableToLiteral,
            LeftPropertyName = "PlateNumber",
            ComparisonOperator = LogicOperator.Equal,
            ExpectedValue = "123-abc"
        },
        new Rule
        {
            RuleType = RuleType.VariableToLiteral,
            LeftPropertyName = "Transmission",
            ComparisonOperator = LogicOperator.NotEqual,
            ExpectedValue = "2345678"
        },
        new Rule
        {
            RuleType = RuleType.VariableToLiteral,
            LeftPropertyName = "Miles",
            ComparisonOperator = LogicOperator.GreaterThan,
            ExpectedValue = "1000"
        }
    };
    RuleSet ruleSet = new RuleSet(ruleList);

    RuleValidationService target = new RuleValidationService();
    target.IsSatisfied(vehicle, ruleSet);
    bool expected = true;
    bool actual;
    actual = target.IsSatisfied(vehicle, ruleSet);
    Assert.AreEqual(expected, actual);
}
```

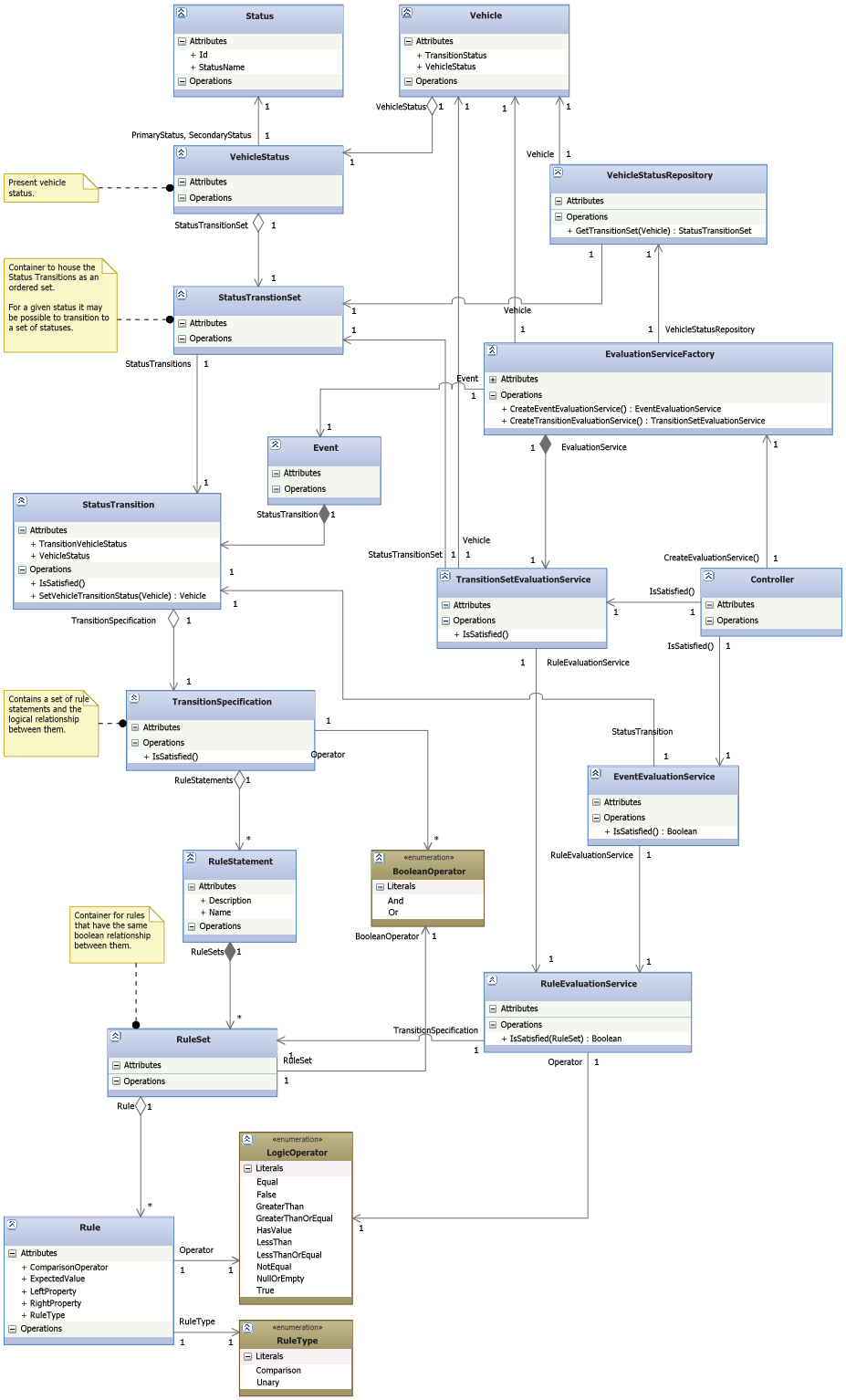
Unit Test of RuleValuationService with AND operation

Listed below is code that can perform the AND operation against the rules in the rule set. The `IsSatisfied()` method unpacks the rule set, builds a expression tree against the vehicle and performs an AND operation between the results of the rules via the Linq `All()` method.

```
public class RuleValidationService
{
    public bool IsSatisfied(Vehicle vehicle, RuleSet ruleSet)
    {
        var rules = ruleSet.Rules;
        var compiledRules = rules.Select(r => CompileRule<Vehicle>(r)).ToList();
        return compiledRules.All(rule => rule(vehicle));
    }

    static Func<T, bool> CompileRule<T>(Rule r)
    {
        var paramExprNode = Expression.Parameter(typeof(T));
        Expression expr = BuildExpr<T>(r, paramExprNode);
        return Expression.Lambda<Func<T, bool>>(expr, paramExprNode).Compile();
    }

    static Expression BuildExpr<T>(Rule r, ParameterExpression paramExprNode)
    {
        var left = MemberExpression.Property(paramExprNode, r.LeftPropertyName);
        var leftPropertyType = typeof(T)
                               .GetProperty(r.LeftPropertyName)
                               .PropertyType;
        ExpressionType tBinary;
        //Is operator a known .NET operator
        if (ExpressionType.TryParse(r.ComparisonOperator, out tBinary))
        {
            var right = Expression.Constant(Convert.ChangeType(
                                            r.ExpectedValue, leftPropertyType));
            return Expression.MakeBinary(tBinary, left, right);
        }
        else
        {
            var method = leftPropertyType.GetMethod(r.ComparisonOperator);
            var tParam = method.GetParameters()[0].ParameterType;
            var right = Expression.Constant(Convert.ChangeType(
                                                r.ExpectedValue, tParam));
            // Use a method call, e.g. 'Contains' n.tag.Contains(something)
            return Expression.Call(left, method, right);
        }
    }
}
```

Proof of Concept: RuleEvaluationEngine code to handle AND operation between Rules

Complete Rules Engine Class Diagram