.NET Business Rules Engine

http://nxbre.org

NxBRE

David Dossot, Project Lead

Version 2.5.1
May 20, 2006

Table of Contents

1. Introduction

1.1. What is NxBRE?

NxBRE is the first open-source rule engine for the .NET platform and a lightweight Business Rules Engine (aka Rule-Based Engine) that offers two different approaches:

- the Flow Engine, which uses XML as a way to control process flow for an application in an external entity. It is basically a wrapper on C#, as it offers all its flow control commands (if/then/else, while, foreach), plus a context of business objects and results. It is a port of **JxBRE** v1.7.1 (SourceForge Project from **Sloan Seaman**) to .NET's Visual C#.

- the Inference Engine, which is a forward-chaining (data driven) deduction engine and that supports concepts like Facts, Queries and Implications (as defined in RuleML Datalog) and like Rule Priority, Mutual Exclusion and Precondition (as found in many commercial engines). It is designed in a way that encourages the separation of roles between the expert who designs the business rules and the programmer who binds them to the business objects.

NxBRE's interest lies first into its simplicity, second in the possibility of easily extending its features by delegating to custom code in the Flow Engine or by writing custom RuleBase adapters or Business Objects binders in the Inference Engine.

NxBRE can be really useful for projects that have to deal with:

- complex business rules that can not be expressed into one uniform structured manner but require the possibility to have free logical expressions,

- changing business rules that force recompilation if the new rules must meet unexpected requirements.

NxBRE is released under LGPL license in order to allow users to legally build commercial solutions that embed **NxBRE**.

The main contributors are:

- **David Dossot, Project Lead**
  Original port, Flow Engine Improvements, Inference Engine, PDF Documentation
- **André Weber, Ron Evans**
  Human Readable Format parser
- ***Stéphane Joyeux***
  XML Schema data type support

Other contributors are acknowledged in the source code, wherever their contributions have been included.

For comments or questions use the SourceForge forums or write to: contact@nxbre.org

1.2. What is this documentation?

This documentation presents the general concepts in **NxBRE**, the main interfaces and classes and the syntax of the XML rule files.

If you are looking for source code and rules samples, check first the regression test files (both C# source code and the related rule files), which demonstrate all the features of **NxBRE**, then look at the provided examples for more information.

1.3. Release Notes

The release notes can be found in the **readme.txt** file included in all **NxBRE** archives.

1.4. Content of the Rulefiles folder

Please check the “readme” file in this folder.

1.5. How to choose between the Flow and the Inference Engine ?

This question comes regularly so it probably deserves a little paragraph here.

To select between the Flow Engine and the Inference Engine, consider the following differences:

- the Inference Engine supports priority, mutual exclusions and pre-conditions,
- the Inference Engine uses a "standard" rule format (RuleML),
- the Inference Engine has an elaborated memory model with support for isolated deduction space.

If any of these things is important for you, you can consider using the Inference Engine ; else stick to the Flow Engine.

Moreover, the Inference Engine is well suited for knowledge bases and expert systems where facts are important to keep and persist because they represent knowledge.

The Flow Engine is really an instantaneous traversal of logical branches using transient data for evaluations of boolean expressions: if you think in terms of “if, then else, while” then you surely want to go for the Flow Engine.

2. The Flow Engine

2.1. Introduction

The Flow Engine of **NxBRE** is controlled by one XML file that contains instructions of three main kinds: rules, logic tests and structure.

For the Flow Engine a rule is not an implication of some sort, like if-then, but a "value object" that implements `IBRERuleFactory`[1], identified by a unique id, and whose type is either an helper object in the assembly or a delegate to a custom piece of code.

The important method in the `IBRERuleFactory` interface is `ExecuteRule` : this method is called by the engine when it is time for the rule to compute its value, i.e. when the execution flow has been directed to hit a rule element in the XML file.

In the same way, operators are defined by objects in the assembly, referenced by their fully qualified names, which implement `IBREOperator`. The important method in the `IBREOperator` interface is `ExecuteComparison` : this method is called by the engine whenever it needs to perform a logical comparison.

Programmers are free to create their own implementations of these interfaces. The Flow Engine comes complete with a reference implementation that provides helper objects implementing these interfaces (chapter 2.3.3).

The engine itself is also defined by an interface (`IFlowEngine`, detailed in the next chapter) that is implemented in **NxBRE** one particular Flow Engine:

- the **Rule Interpreter** that is the slowest and safest one: the XML rule file is parsed each time a process is launched and each rule is interpreted when they are read,

For a long period of time, the following implementation has been foreseen but never realized:

- *the **Rule Compiler** that performs an upfront transformation of the rule file into C#, compiles it and then executes it every time processing is called.*

The initial performance tests were showing an performance improvement of one order of magnitude.

1 *Though IBRERule would have been a better name, the original JxBRE name has been kept (I has been added).*

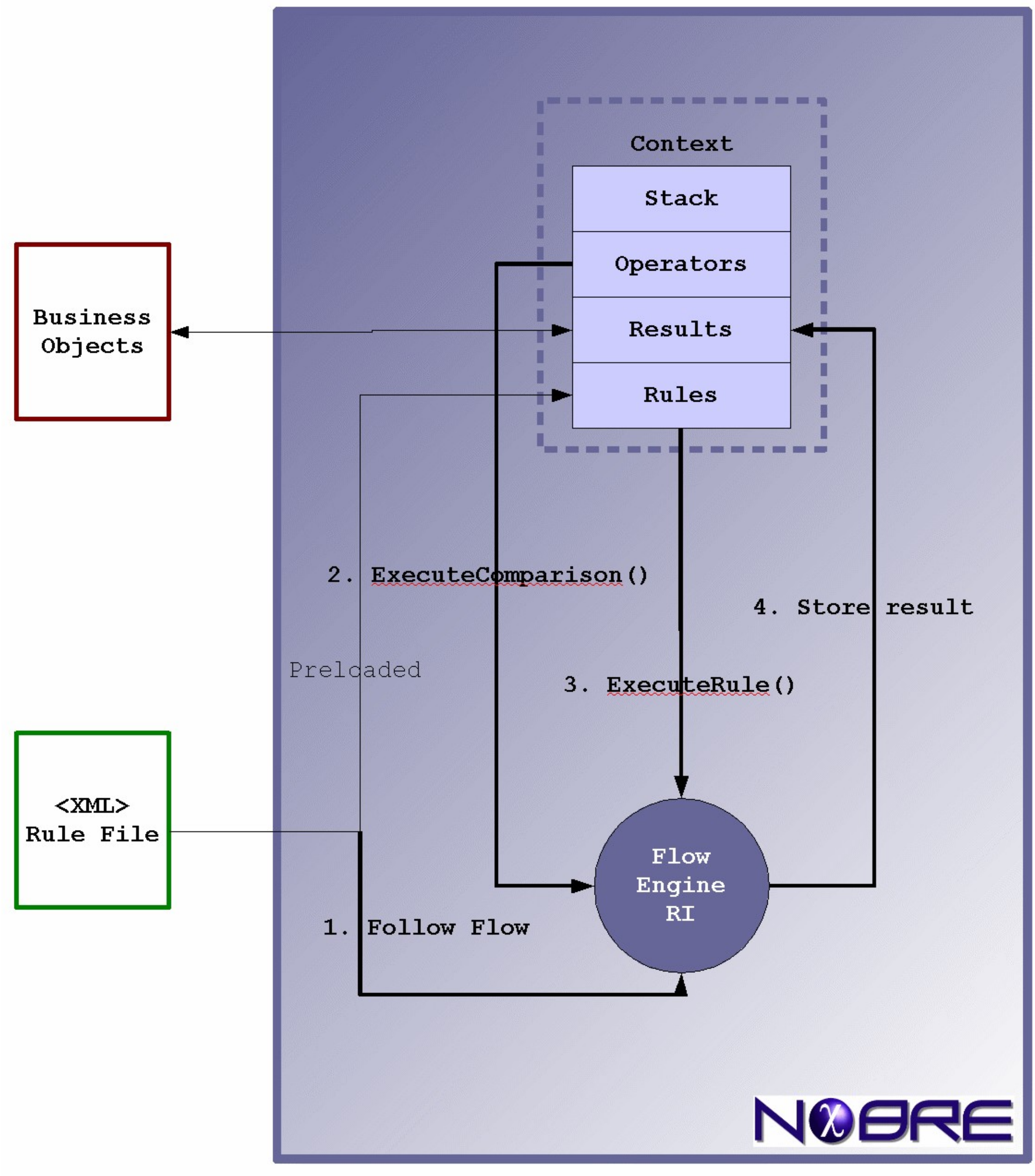
2.2. The IFlowEngine interface

As explained before, the Flow Engine is defined by an interface named `IFlowEngine`, which also implements several other interfaces. These are detailed in the following table.

<table>
<tr><th>Method, Property or Event</th><th>Package.Interface</th><th>Description</th></tr>
<tr><td>Init</td><td>net.ideaity.util.
IInitializable</td><td>General definition of an initializable object, basically one that has an Init method with one object parameter.</td></tr>
<tr><td>DispatchLog</td><td>net.ideaity.util.events.
IlogDispatcher</td><td>Called when a log is generated by the engine (logs are categorized by a priority level).</td></tr>
<tr><td>DispatchException</td><td>net.ideaity.util.events.
IexceptionDispatcher</td><td>Called when an exception is thrown by the engine (exceptions are categorized by a severity level).</td></tr>
<tr><td>DispatchRuleResult</td><td>org.nxbre.rule.
IBREDispatcher</td><td>Called when a result is added to the context.</td></tr>
<tr><td>RuleContext</td><td rowspan="7">org.nxbre.
IFlowEngine</td><td>The engine's context.</td></tr>
<tr><td>XmlDocumentRules</td><td>The loaded XML rules.</td></tr>
<tr><td>Process</td><td>Start the engine and process all accessible rules.</td></tr>
<tr><td>Process(ruleSetId)</td><td>Start the engine and process all rules in a set.</td></tr>
<tr><td>Stop</td><td>Stop the engine abruptly.</td></tr>
<tr><td>Reset</td><td>Place the engine in a state where previous results are cleared, ready to process again.</td></tr>
<tr><td>Clone</td><td>System.
ICloneable</td><td>Returns a clone of the engine, ready to process. The cloning depth should not extend to the objects in the context.</td></tr>
</table>

Please refer to chapter 4 for the detailed `org.nxbre` API.

2.3. Rule Interpreter Implementation: org.nxbre.ri[2]



2 *Note that RI has been re-interpreted from Reference Implementation to Rule Interpreter after the introduction of the Rule Compiler Implementation concept...*

2.3.1. Execution

NxBRE uses a context object that is used to carry information about its execution environment, which are:

- the available operators,
- the loaded rules,
- the user's business objects and the generated results,
- and a stack trace.

When parsing the rule file at initialization time, the engine loads all the rule objects in the context. At execution time, the engine follows the execution flow defined by tests and loops. When it reaches a rule element, it gets the corresponding object by its id and calls the `ExecuteRule` method on it. It then stores the result of this call in the context result.

The `ExecuteRule` method provides the callee with the **NxBRE**'s context, a map of the optional additional parameters that could have been provided in the XML file and a third parameter (Step) which can also be defined in the XML file.

User's business objects are placed before execution in the context result pool where they are accessible to the engine. After execution, these objects might have been modified, new ones might have been created in the result context (asserted) and some might have been removed from there (retracted).

2.3.2. Engine

The Rule Interpreter engine `org.nxbre.ri.BREImpl` has the following characteristics:

- it can be initialized either by an `XPathDocument`, or by rule driver that is responsible for fetching rules from a specific source (see chapter 2.3.4),

- it will not break on exceptions, so it is of programmer's responsibility to stop it, if it is necessary,

- it is thread safe, as long as each thread uses a clone.

2.3.3. Helper Objects

The following table presents different rules helpers in the Rule Interpreter implementation.

Class name in: org.nxbre.ri.helpers.rule	**Description**
`Decrement`*	`Integer that decrements.`
`Exception`	`Raises an exception.`
`False`	`Constant boolean false.`
`FatalException`	`Raises a fatal exception.`
`Increment`*	`Integer that increments.`

* This helper is stateful therefore not to be used in a multi-threaded approach.

Class name in: org.nxbre.ri.helpers.rule	**Description**
IncrementInit*	Incrementor or Decrementor reset.
ObjectLookup	Reflection call on a class or an object.
True	Constant boolean true.
Value	Instantiation of any type.

The following table presents different operators helpers in the Rule Interpreter implementation.

Class name in: org.nxbre.ri.helpers.operators	**Description**
Equals	**==**
GreaterThan	**>**
GreaterThanEqualTo	**>=**
InstanceOf	**InstanceOf || SubtypeOf**
LessThan	**<**
LessThanEqualTo	**<=**
NotEquals	**!=**

2.3.4. Rule Drivers

The following table presents the different drivers in the Rule Interpreter implementation. They are able to read rules from any URI (file system or URL).

Class name in: org.nxbre.ri.drivers	**Description**
BusinessRulesFileDriver	Loads up rules in the *native* format, i.e. defined by businessRules.xsd (see chapter 2.4).
XSLTRulesFileDriver	Loads rules in any custom format and transform it to the native format by performing an XSLT.
XBusinessRulesFileDriver	Specialization of the previous driver that transforms rules defined by xBusinessRules.dtd (see chapter).

2.3.5. Factories

To facilitate the instantiation and initialization of the engine, a few factories are available. After instantiation they return an object implementing `IFlowEngine` each time `NewBRE` is called.

* This helper is stateful therefore not to be used in a multi-threaded approach.

They are presented in the following table.

Class name in: org.nxbre.ri.factories	Description
BREFactory	Instantiate an engine with a specific driver and optional event handlers.
BREFactoryConsole	Specialization of the previous factory that registers handlers that write their events to the console.
BRECloneFactory	Singleton-like instantiation of an engine that returns a different clone each times (useful for a multi-threaded environment).

2.3.6. Threading Model

Since the Rule Interpreter implementation is not knowledge-base oriented but flow-oriented, it does not make sense to share the context between several engines. It is why the recommended multi-threaded approach is to execute once (and discard after usage) clones of a preloaded engine and using only stateless helpers (see chapter 2.3.3).

Consequently, the Rule Interpreter implementation is voluntarily not based on synchronized collections (the context members). Should you need to use the same context in different threads, serialize the calls in a synchronized calling method. Trying to synchronize the core objects will expose the programmer to the difficulties of sharing the context amongst concurrently executing flow engines.

2.4. Rule Files Formats

With this document, you should find pictures representing the XML Schemas of the two formats supported by the Rule Interpreter implementation:

- businessRules.xsd.png: the native syntax, coming from **JxBRE**,
- xBusinessRules.xsd.png: the extended syntax,

The new rule format has been introduced in order to solve the main issues of businessRules.xsd:

- direct references made to fully qualified class names of **NxBRE** and .NET,
- poor grammar leading to a confusing syntax.

The following tables show the details of the matching between the new syntax and the native one.

xBusinessRules Element	**businessRules Element**	**Factory(ies) in org.nxbre.ri.helpers.operators**
Between	Compare	LessThan, LessThanEqualTo, GreaterThan, GreaterThanEqualTo
In		Equals
IsAsserted		InstanceOf
IsTrue		Equals
IsFalse		Equals
Equals		Equals
NotEquals		NotEquals
InstanceOf		InstanceOf
LessThan		LessThan
LessThanEqualTo		LessThanEqualTo
GreaterThan		GreaterThan
GreaterThanEqualTo		GreaterThanEqualTo

xBusinessRules Element	businessRules Element	Factory in org.nxbre.ri.helpers.rules	Type
Boolean	Rule	Value	System.Boolean
Byte			System.SByte
Date DateTime Time			System.DateTime
Decimal			System.Decimal
Double			System.Double
Integer			System.Int32
Long			System.Int64
Exception			System.Exception
Short			System.Int16
Single			System.Single
String			System.Double

xBusinessRules Element	businessRules Element	Factory in org.nxbre.ri.helpers.rules
True	Rule	org.nxbre.ri.helpers.rule.True
False		org.nxbre.ri.helpers.rule.False

xBusinessRules Element	businessRules Element
Assert	org.nxbre.ri.helpers.rule.Value
Evaluate	org.nxbre.ri.helpers.rule.Value
Modify	org.nxbre.ri.helpers.rule.Value
ObjectLookup	org.nxbre.ri.helpers.rule.ObjectLookup
Increment*	org.nxbre.ri.helpers.rule.Increment
(not kept as it is simply a negative Increment)	org.nxbre.ri.helpers.rule.Decrement
ThrowException	org.nxbre.ri.helpers.rule.Exception
ThrowFatalException	org.nxbre.ri.helpers.rule.FatalException

xBusinessRules Element	businessRules Element	Type
And	Condition	AND
Or		OR
Not		NOT

The following elements are similar in both grammars :

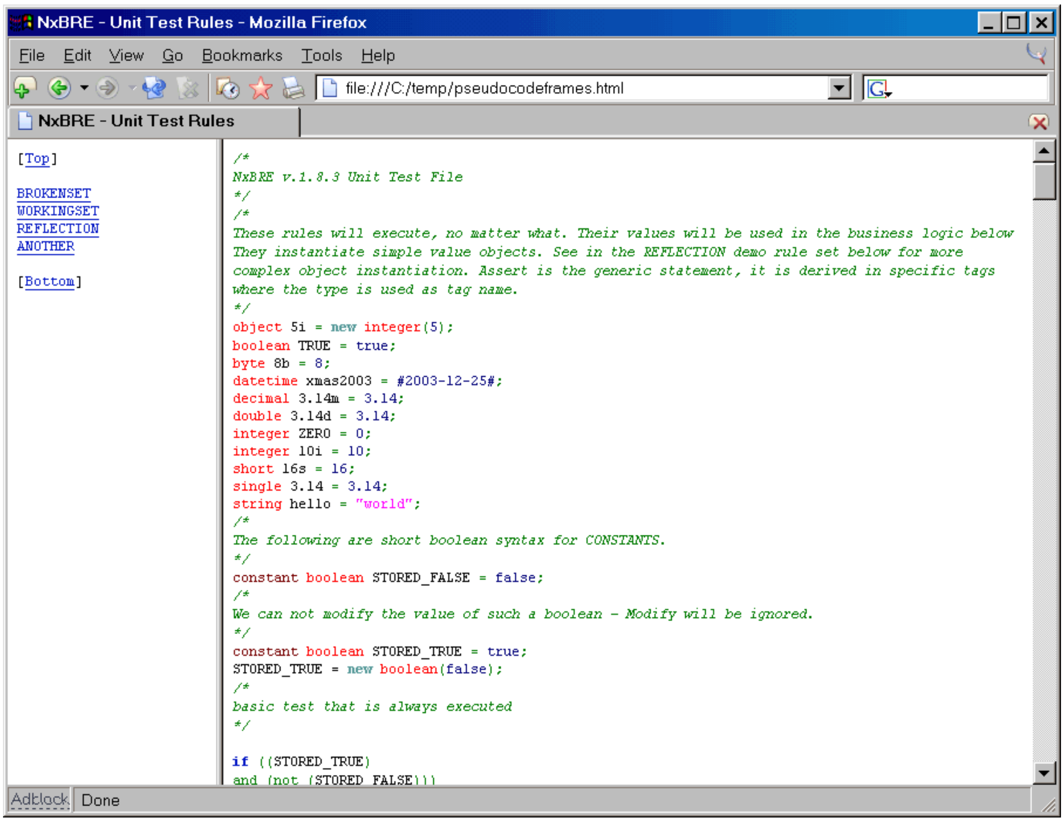
InvokeSet ForEach Log Logic Retract Set While

* This helper is stateful therefore not to be used in a multi-threaded approach.

2.4.1. Pseudo-code Rendering

A utility class named `PseudoCodeRenderer` allows rules files that validate on xBusinessRules.xsd to be rendered as pseudo-code HTML files.

The pseudo-code syntax used is somewhere between C# and Java, the main goal being to facilitate the review of long and complex rules files by transforming highly hierarchical XML files into a view that fits more the programmer's standards.



The renderer can generate the 3 HTML documents used to build the above view :

- the index on the left,
- the body on the right,
- and the frame set to bind them all.

2.4.2. Rules Format Comparison

The next 3 pages are designed to allow the comparison of the same rules expressed in respectively native format, extended format and pseudo-code.

```xml
<BusinessRules xsi:noNamespaceSchemaLocation="http://nxbre.org/businessRules.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Rule id="10i" factory="org.nxbre.ri.helpers.rule.Value">
        <Parameter name="Value" value="10"/>
        <Parameter name="Type" value="System.Int32"/>
    </Rule>
    <Rule id="40i" factory="org.nxbre.ri.helpers.rule.Value">
        <Parameter name="Value" value="40"/>
        <Parameter name="Type" value="System.Int32"/>
    </Rule>
    <Rule factory="org.nxbre.ri.helpers.rule.ObjectLookup" id="QuantityOrdered">
        <Parameter name="ObjectId" value="CurrentOrder"/>
        <Parameter name="Member" value="Quantity"/>
    </Rule>
    <Logic>
        <If>
            <Condition type="AND">
                <Compare leftId="ClientRating" operator="org.nxbre.ri.helpers.operators.GreaterThanEqualTo" rightId="ClientRatingThreshold">
                    <Rule factory="org.nxbre.ri.helpers.rule.ObjectLookup" id="ClientRating">
                        <Parameter name="ObjectId" value="CurrentOrder"/>
                        <Parameter name="Member" value="ClientRating"/>
                    </Rule>
                    <Rule id="ClientRatingThreshold" factory="org.nxbre.ri.helpers.rule.Value">
                        <Parameter name="Value" value="C"/>
                        <Parameter name="Type" value="System.String"/>
                    </Rule>
                </Compare>
            </Condition>
            <Do>
                <!-- Discount rules for high rate customers -->
                <Logic>
                    <If>
                        <Condition type="AND">
                            <Compare leftId="QuantityOrdered" operator="org.nxbre.ri.helpers.operators.GreaterThan" rightId="40i"/>
                        </Condition>
                        <Do>
                            <Rule id="AppliedDiscount">
                                <Parameter name="Type" value=""/>
                                <Parameter name="Percent" value=".7"/>
                            </Rule>
                        </Do>
                    </If>
                    <ElseIf>
                        <Condition type="AND">
                            <Compare leftId="QuantityOrdered" operator="org.nxbre.ri.helpers.operators.GreaterThan" rightId="10i"/>
                        </Condition>
                        <Do>
                            <Rule id="AppliedDiscount">
                                <Parameter name="Type" value=""/>
                                <Parameter name="Percent" value=".8"/>
                            </Rule>
                        </Do>
                    </ElseIf>
                    <Else>
                        <Rule id="AppliedDiscount">
                            <Parameter name="Type" value=""/>
                            <Parameter name="Percent" value=".9"/>
                        </Rule>
                    </Else>
                </Logic>
            </Do>
        </If>
        <Else>
            <!-- Discount rules for low rate customers -->
            <Logic>
                <If>
                    <Condition type="AND">
                        <Compare leftId="QuantityOrdered" operator="org.nxbre.ri.helpers.operators.GreaterThan" rightId="40i"/>
                    </Condition>
                    <Do>
                        <Rule id="AppliedDiscount">
                            <Parameter name="Type" value=""/>
                            <Parameter name="Percent" value=".9"/>
                        </Rule>
                    </Do>
                </If>
                <Else>
                    <Rule id="AppliedDiscount">
                        <Parameter name="Type" value=""/>
                        <Parameter name="Percent" value="1"/>
                    </Rule>
                </Else>
            </Logic>
        </Else>
    </Logic>
</BusinessRules>
```

```xml
<xBusinessRules   xsi:noNamespaceSchemaLocation="http://nxbre.org/xBusinessRules.xsd"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <!-- global values -->
    <Integer id="10i" value="10"/>
    <Integer id="40i" value="40"/>
    <ObjectLookup id="QuantityOrdered" objectId="CurrentOrder" member="Quantity"/>
    <Logic>
        <If>
            <And>
                <GreaterThanEqualTo leftId="ClientRating" rightId="ClientRatingThreshold">
                    <ObjectLookup id="ClientRating" objectId="CurrentOrder" member="ClientRating"/>
                    <String id="ClientRatingThreshold" value="C"/>
                </GreaterThanEqualTo>
            </And>
            <Do>
                <!-- Discount rules for high rate customers -->
                <Logic>
                    <If>
                        <And>
                            <GreaterThan leftId="QuantityOrdered" rightId="40i"/>
                        </And>
                        <Do>
                            <Evaluate id="AppliedDiscount">
                                <Parameter name="Percent" value=".7"/>
                            </Evaluate>
                        </Do>
                    </If>
                    <ElseIf>
                        <And>
                            <GreaterThan leftId="QuantityOrdered" rightId="10i"/>
                        </And>
                        <Do>
                            <Evaluate id="AppliedDiscount">
                                <Parameter name="Percent" value=".8"/>
                            </Evaluate>
                        </Do>
                    </ElseIf>
                    <Else>
                        <Evaluate id="AppliedDiscount">
                            <Parameter name="Percent" value=".9"/>
                        </Evaluate>
                    </Else>
                </Logic>
            </Do>
        </If>
        <Else>
            <!-- Discount rules for low rate customers -->
            <Logic>
                <If>
                    <And>
                        <GreaterThan leftId="QuantityOrdered" rightId="40i"/>
                    </And>
                    <Do>
                        <Evaluate id="AppliedDiscount">
                            <Parameter name="Percent" value=".9"/>
                        </Evaluate>
                    </Do>
                </If>
                <Else>
                    <Evaluate id="AppliedDiscount">
                        <Parameter name="Percent" value="1"/>
                    </Evaluate>
                </Else>
            </Logic>
        </Else>
    </Logic>
</xBusinessRules>
```

```
/*
global values
*/
integer 10i = 10;
integer 40i = 40;
object QuantityOrdered = CurrentOrder.Quantity;

if ((object ClientRating = CurrentOrder.ClientRating) >= (string ClientRatingThreshold = "C"))
{
        /*
        Discount rules for high rate customers
        */

        if (QuantityOrdered > 40i)
        {
        evaluate(AppliedDiscount(Percent = ".7"));
        }
        elseif (QuantityOrdered > 10i)
        {
        evaluate(AppliedDiscount(Percent = ".8"));
        }
        else
        {
        evaluate(AppliedDiscount(Percent = ".9"));
        }

}
else
{

        /*
        Discount rules for low rate customers
        */

        if (QuantityOrdered > 40i)
        {
        evaluate(AppliedDiscount(Percent = ".9"));
        }
        else
        {
        evaluate(AppliedDiscount(Percent = "1"));
        }

}
```

3. The Inference Engine: org.nxbre.ie

3.1. RuleML Naf Datalog Concepts

The inference engine has been developed using RuleML Naf Datalog[3] as a conceptual model.

Please visit http://www.ruleml.org for more information.

*NB. RuleML is constantly evolving. When stabilized on the official version 1.0, a big effort will be made to allow **NxBRE** to support it ; or, at least, to replace the **NxBRE** specific sub-constructs into normalized ones (for ex. Mutex, Expressions...).*

3.1.1. Atoms

An atom is a named relationship between predicates, either individual (fixed values) or variable ones (value placeholders). Atoms can not be used on their own: they must be used in facts, queries or implications.

In queries and implications, atoms are the “listening patterns” that select values from the fact base. The pattern matching is based on the relationship name and the number, type and position of predicates.

RuleML	Natural Language

`<Atom>`
 `<op>`
  `<Rel>luxury</Rel>`
 `</op>`
 `<Var>product</Var>`
`</Atom>`
	Select all products that are considered luxury.

NxBRE can also recognize function predicates by analyzing the values of individual predicates (see chapter 3.4.3).

An atom can be made negative if it is surrounded by naf (negation as failure).

RuleML	Natural Language

`<Naf>`
 `<Atom>`
  `<op>`
   `<Rel>luxury</Rel>`
  `</op>`
  `<Var>product</Var>`
 `</Atom>`
`</Naf>`
	Be positive if no products are considered luxury.
	
	NB. This negative atom will never produce any value in a query or an implication. For example, it does not select regular products!

NB. A negative atom never produces any value : this must be taken in account when designing queries and implications...

3 Currently RuleML Naf Datalog version 0.86 and 0.9 are partially supported.

3.1.2. Logical Operators

NxBRE supports the *and* and *or* logical operators. It must be very clear for the user that these operators does not simply manipulate boolean values but also data emerging from the atoms they link.

Moreover *and* groups can perform combination of atoms, as shown in the “sibling” implication of the Gedcom rule base. By using different variable names (child1 and child2) a the same position in the atom, this asks the engine to look for combination of different facts that satisfy the *and* group.

3.1.3. Facts

A fact is an assertion about something that is true. It is a special atom that contains only individual predicates.

RuleML	Natural Language

`<Atom>`
` <op>`
`  <Rel>luxury</Rel>`
` </op>`
` <Ind>Porsche</Ind>`
`</Atom>`
	`Porsche is luxury.`

NB. The working memory of **NxBRE** can only contain one instance of a particular fact.

3.1.4. Queries

A query is a way of expressing a question that will be asked to the fact base. It is a group of atoms that are not facts, linked together by a logical operator (and / or).

RuleML	Natural Language

`<Query>`
` <body>`
`  <Atom>`
`   <op>`
`    <Rel>discount</Rel>`
`   </op>`
`   <Var>customer</Var>`
`   <Var>product</Var>`
`   <Var>amount</Var>`
`  </Atom>`
` </body>`
`</Query>`
	`Give the discount amounts for all customers buying any products.`

RuleML	Natural Language

`<Query>`
` <body>`
`  <And>`
`   <Atom>`
`    <op>`
`     <Rel>childIn</Rel>`
`    </op>`
`    <Var>child1</Var>`
`    <Var>family</Var>`
`   </Atom>`
`   <Atom>`
`    <op>`
`     <Rel>childIn</Rel>`
`    </op>`
`    <Var>child2</Var>`
`    <Var>family</Var>`
`   </Atom>`
`  </And>`
` </body>`
`</Query>`
	Give pairs of different children in the same family.

NB. The *m* results of a query containing *n* atoms in *and* group, will be an array of facts with *m* rows and *n* columns. With *or*, the number of columns will vary. Same when using negative atoms or function based atom relations (see chapter 3.3.5).

3.1.5. Implications

An implication is a query whose results will be used to assert new facts. The new facts will be created by using the head atom as a template and the values of the variable parts of the atoms of the query to populate the variables of this template.

RuleML	Natural Language

`<Implies>`
` <head>`
`  <Atom>`
`   <op>`
`    <Rel>premium</Rel>`
`   </op>`
`   <Var>customer</Var>`
`  </Atom>`
` </head>`
` <body>`
`  <Atom>`
`   <op>`
`    <Rel>spending</Rel>`
`   </op>`
`   <Var>customer</Var>`
`   <Ind>min(5000,euro)</Ind>`
`   <Ind>previous year</Ind>`
`  </Atom>`
` </body>`
`</Implies>`
	A customer is premium if their spending has been min 5000 euro in the previous year.

If the head part does not contain any variable predicate, the fact will be asserted if the body part returns at least one result.

Unless the engine has been set to enforce strict implications, **NxBRE** will ignore the assertion attempt of a fact that is not completely resolved, i.e. If the body part has not produced enough data in a result to populate all the variable parts of the template atom of the head part.

In **NxBRE**, advanced concepts like priority, mutual exclusion and pre-condition have been introduced, allowed an even finer translation of business rules into RuleML.

3.2. What is new in version 2.5?

Many new features and elements have been added to RuleML 0.9 Naf Datalog sub-language but not all of them have been introduced to **NxBRE** so far. This said, what have been added is already a significant step and is worth detailing here.

3.2.1. Slots

Slots have been introduced in RuleML 0.86 but their support has been delayed until **NxBRE** 2.5. They are a convenient way of naming an element, which allows an easy retrieval in the application.

RuleML	Description
<Atom> <op> <Rel>bonus</Rel> </op> <Var>employee</Var> <slot> <Ind>amount</Ind> <Ind uri="nxbre://expression"> {var:score}*3 </Ind> </slot> </Atom>	A slot contains exactly two child elements: the first one is the slot name and the second one is the slot value. Only individual are allowed for slot names ; any supported elements are allowed for slot values (Ind, Var...).

Getting the value of the slotted predicate would then be possible via this simple code:

```
myAtom.GetPredicateValue("amount");
```

Since version 2.5.1, slots can also be used to contribute named value to the deduction part of the implication (head), much like variables do.

RuleML	Description
<Implies> <Atom> <Rel>measure</Rel> <slot> <Ind>**amount**</Ind> <Ind uri="nxbre://operator">GreaterThan(25)</Ind> </slot> </Atom> <Atom> <Rel>warning</Rel> <Var>**amount**</Var> </Atom> </Implies>	Using a slot allows a comparison made at a predicate level to produce a named value, which is a very efficient and compact syntax. Named values are directly retrieved with variable predicates in the deduction part of the implication.

3.2.2. Multi-syntax

RuleML 0.9 has introduced a flexible schema that allows the support for optional elements. This has allowed to define three syntaxes: compact, standard and expanded.

To discover how these syntaxes differ, the best is to compare the same rule base saved under the three formats. The distribution of **NxBRE** contains the following sample rule bases: *own_expanded.ruleml*, *own.ruleml* and *own_compact.ruleml* that demonstrate the three syntaxes.

The *compact* syntax removes all optional elements (named with lower case, except *slot*) and assumes that the children of an implication are in the body/head (if/then) order. This is compulsory because *body* and *head* are optional elements, thus are not present in the compact syntax.

The *standard* syntax uses only *body*, *head* and *op* optional elements.

The *expanded* syntax adds several optional elements, whose most notable is *arg* which allows to position atom members with a numeric index, starting at 1.

3.2.3. *Typed data*

Support for XML Schema data types have been introduced, which allows a clean support of typed data: XML editors provide immediate feedback if a data is not correct. This also allows the creation of typed facts directly from rule bases, without using slow and error prone C# expressions.

RuleML 0.86 + C# expression

```
<fact>
  <_head>
    <atom>
      <ind>expr:System.DateTime.Parse("1999/12/31 23:59:59")</ind>
    </atom>
  </_head>
</fact>
```

Same fact in RuleML 0.9

```
<Atom>
  <Data xsi:type="xs:dateTime">1999-12-31T23:59:59Z</Data>
</Atom>
```

If **NxBRE** loads this kind of typed data, it will save it as typed data. Facts asserted in memory or loaded from *Ind* elements do not carry a XML Schema type and will not be saved as typed data unless the “Force Data Typing” save attribute is used when instantiating the adapter.

3.2.4. *Deterministic resolution*

The *uri* attribute that has been introduced in RuleML 0.9 has been leveraged to replace prefix-based content resolution with a deterministic one.

Type	RuleML 0.86 + Prefix	RuleML 0.9
Operator	`<ind>NxBRE:Equals(100)</ind>`	**`<Ind uri="nxbre://operator">Equals(100)</Ind>`**
Expression	`<ind>expr:{ind}.StartsWith("hello")</ind>`	**`<Ind uri="nxbre://expression">{ind}.StartsWith("hello")</Ind>`**
Binder Resolved Expression	`<ind>binder:CalculateTotalWeight</ind>` *or:* *<ind>CalculateTotalWeight()</ind>* *for regular expression based recognition.*	**`<Ind uri="nxbre://binder">CalculateTotalWeight</Ind>`** **NB. The regular expression is used but only for parameters extraction.**
Function Based Atom Relation	`<rel>expr:{var:Date} < System.DateTime.Now</rel>`	**`<Rel uri="nxbre://expression">{var:Date} < System.DateTime.Now</Rel>`**
Binder Resolved Atom Relation	`<rel>binder:WithinTolerance</rel>`	**`<Rel uri="nxbre://binder">WithinTolerance</Rel>`**

3.2.5. Equivalence

It is possible to define two atoms as equivalent, which is a very useful feature for reducing the size of queries or implications.

RuleML	Natural Language
<Equivalent> <oid> <Ind>Own/belong equivalence</Ind> </oid> <Atom> <Rel>own</Rel> <Var>person</Var> <Var>stuff</Var> </Atom> <Atom> <Rel>belongs</Rel> <Var>stuff</Var> <Var>person</Var> </Atom> </Equivalent>	Saying that a person owns a stuff is equivalent to saying that the stuff belongs to the person.

If more *n* atoms are equivalent, *n-1* equivalent pairs are needed, as **NxBRE** explores the equivalence graph.

It is not necessary that the atoms in the equivalence pairs have their variables named identically with the atoms variables in the queries and implications: **NxBRE** recognizes atoms on the relation type, number of members and equal individual/data values.

For example, the following atom:

```
<Atom>
 <Rel>own</Rel>
 <Var>client</Var>
 <Var>object</Var>
</Atom>
```

will be automatically translated into its equivalent:

```
<Atom>
 <Rel>belongs</Rel>
 <Var>object</Var>
 <Var>client</Var>
</Atom>
```

and the system will search matching facts for both.

When translating the body clause with equivalent atoms, **NxBRE** takes in account the current logical operator and if the atom is in a Naf block. This means that in an Or block, negative atoms are surrounded by And, while positive ones are directly inserted. And in an And block, positive atoms are surrounded by Or, while negative ones are directly inserted.

3.2.6. Integrity protection

It is possible to write special queries dedicated to verify the integrity of the rule base, more precisely the facts present in the current working memory at the end of the inference process.

If this query does not return any row, **NxBRE** will throw an *org.nxbre.ie.IntegrityException*. Note that if the query returns empty row(s), which is possible when using Naf or function based relations, the system will not throw an exception. This means that the integrity queries must be written in a way that they return zero row if the rule base integrity has been violated.

RuleML

```
<Protect>
 <Integrity>
  <oid>
   <Ind>An object can not be gold and rusty</Ind>
  </oid>
  <Or>
   <And>
    <Atom>
     <Rel>gold</Rel>
     <Var>object</Var>
    </Atom>
    <Naf>
     <Atom>
      <Rel>rusty</Rel>
      <Var>object</Var>
     </Atom>
    </Naf>
   </And>
   <And>
    <Naf>
     <Atom>
       <Rel>gold</Rel>
      <Var>object</Var>
     </Atom>
    </Naf>
    <Atom>
     <Rel>rusty</Rel>
     <Var>object</Var>
    </Atom>
   </And>
   <And>
    <Naf>
     <Atom>
      <Rel>gold</Rel>
      <Var>object</Var>
     </Atom>
    </Naf>
    <Naf>
     <Atom>
      <Rel>rusty</Rel>
      <Var>object</Var>
     </Atom>
    </Naf>
   </And>
  </Or>
 </Integrity>
</Protect>
```

This example enforces the fact that an object can not be gold and rusty, so it is either “gold and not rusty”, “not gold and rusty” or “neither gold nor rusty”.

3.2.7. Migrating from a previous version

The simplest way of converting a rule base is to use the **Inference Engine Console** to load in one format and save in the other.

Note that the Console saves in standard syntax and does not force the data typing.

It is also very important to note that if the rule base uses a binder to resolved expressions, functions or formulas, the binder must have been loaded as well. Failing to load a required binder will end up with a loss of information in the saved rule base.

Consequently, to have full control on the save options, the best is to use a code similar to this one:

```
IInferenceEngine ie = new IEImpl(binder);
ie.LoadRuleBase(new RuleML086NafDatalogAdapter(oldRules, FileAccess.Read));
ie.SaveRuleBase(new RuleML09NafDatalogAdapter(newRules, FileAccess.Write,
                        SaveFormatAttributes.Compact |
                        SaveFormatAttributes.ForceDataTyping));
```

3.3. Advanced Concepts

To remain valid with RuleML Datalog Schema (see http://www.ruleml.org), advanced implication parameters can be stored in the optional rule label. This might sound like an heresy but is conform with the original intention:

> *oid is a label for a clause ; it must be individual, this allows naming of a rule in a fashion that is accessible, within the knowledge representation; e.g., this can help for representing prioritization between rules.*

3.3.1. Priority

The priority defines in which order the implications will be evaluated. It is an integer between 0 and 100 (both included), with 0 being the lowest priority.

RuleML	Description

`<Implies>`
` <oid>`
`  <Ind>label:Lower;priority:20</Ind>`
` </oid>`
` <head>`
`  (...)`
	Defines an implication labeled "Lower" whose priority is 20.

If all the implications have the same priority, there is no guarantee on which will be the first evaluated implication.

3.3.2. Mutual Exclusion

The mutual exclusion (aka mutex) represents the fact that if one implication is positive (i.e. its query part returned at least one result, whether the asserted fact was new or not does affect implication positivity), all the implications it mutex-locks will not be evaluated.

Several implications can reciprocally mutex-lock themselves: they then form a mutex chain. The first positive implication will disable all the other implications in the chain. The priority of the implications defines what would be the mutex chain leader.

RuleML	Description

`<Implies>`
` <oid>`
`  <Ind>label:polite;mutex:mundane</Ind>`
` </oid>`
` <head>`
`  (...)`
	Defines an implication labeled "polite" that mutexes an implication labeled "mundane".

`<Implies>`
` <oid>`
`  <Ind>label:mundane;priority:25</Ind>`
` </oid>`
` <head>`
`  (...)`
	Defines an implication labeled "mundane" whose priority is 25.
	
	NB. the mutex with the implication labeled "polite" could have been defined here.

It is very important to understand that the mutex-lock is maintained for the duration of the process cycle (see chapter 3.7). Therefore the granularity of the asserted business objects (see chapter 3.4.3) must be adapted in case mutex are used. In this case, you would assert only the facts that can enter in a mutex lock (for example facts from a single customer) before starting the process cycle. Asserting common facts in the global working memory and related facts in isolated memories would be a good strategy (see chapter 3.6.1).

NB. The provided binding test rule files and related classes demonstrate the problematic of facts granularity.

3.3.3. Pre-Condition

The pre-condition represents the fact that an implication will be evaluated only if another implication was positive before its evaluation in the same process cycle.

Several implications can cascade pre-condition themselves: they then form a pre-condition hierarchy. The position in the hierarchy defines what would be the pre-condition order. Note that the priorities in one hierarchy must be equal or reflect the hierarchy (see chapter 3.3.7).

As for mutual exclusion, the same important remark concerning facts granularity applies to pre-condition locking (see chapter 3.3.2).

RuleML	**Description**

`<Implies>`
 `<oid>`
  `<Ind>label:Rule X;precondition:Rule Z</Ind>`
 `</oid>`
 `<head>`
  `(...)`
	Defines an implication labeled “Rule X” whose evaluation will be done by the engine only if the implication labeled “Rule Z” has been positive in the same process cycle.

3.3.4. Implication Action

The action represents the fact that the implication will either assert, retract or count the facts produced by the inference process, the default being assert.

RuleML	Description
<Implies> <oid> <Ind>label:Rule X;action:retract</Ind> </oid> <head> (...)	Defines an implication labeled "Rule X" whose evaluation will retract facts produced by the inference process.
<Implies> <oid> <Ind>label:Rule Y;action:modify</Ind> </oid> <head> (...)	Defines an implication labeled "Rule Y" whose evaluation will modify the facts found matching the resolved head part of the implication. This means that a query will be dynamically built, based on the deduction atom whose variable predicates would have been resolved with the values coming from the body part. The facts selected by this query will then be modified, with any formula resolved in the context of each of them. The label of the original facts is preserved.
<Implies> <oid> <Ind>label:Rule Z;action:count</Ind> </oid> <head> (...)	Defines an implication labeled "Rule Z" whose evaluation will produce a fact where all the variable predicates will be replaced by the number of results produced by the body part, even if it is 0. Hence it is always positive, and must be used with care.

Whether an implication has asserted, retracted or modified facts, it is considered positive for the mutex and pre-condition mechanisms.

When using the modify action, the implementer must bear in mind the need to stabilize the fact base, i.e. Depending on the modification you perform and the conditions that trigger this modification, it might be needed to implement a specific stop condition to avoid infinite inference loops. For example, if an implication constantly modifies the same fact, the fact base will not stabilize and the engine will keep on inferring until hitting the maximum iteration limit (see chapter 3.7).

3.3.5. Function Based Atom Relations

It is possible to use a function to define and evaluate the relation between the predicates of an atom. This kind of atom does not perform any pattern matching in the fact base, hence can not produce any value in a query or an implication, but are used for evaluating the relation between predicates provided by other atoms in the same logical *and* block.

RuleML	**Description**
<Atom> <op> <Rel uri="nxbre://operator">GreaterThan()</Rel> </op> (...)	Evaluates if the first predicate is greater than the second one, using an operator provided by NxBRE.
<Atom> <op> <Rel uri="nxbre://binder">IsInRange</Rel> </op> **(...)**	Passes the predicates as arguments to the IsInRange custom function defined in the binder associated with the rulebase.
<Atom> <op> <Rel uri="nxbre://expression">({var:X})>=5</Rel> </op> (...)	In this case, it evaluates the C# expression by automatically binding the variable predicates of the atom to the {var:*variable_name*} placeholders of the expression.

NB. Unlike with standard relations, functions relations are evaluated at inference time, leading to more flexibility and less performance.

3.3.6. Formula

A formula is an expression used in the deduction atom of a modifying implication. It allows to compute new predicate values based on the values coming from the query part of the implication and from the current values of the modified fact(s).

RuleML	Description
<Implies> <oid> <Ind>label:update total weight;action:modify</Ind> </oid> <head> <Atom> <op> <Rel>Chocolate_Box_Weight</Rel> </op> <Var>Box</Var> **<Ind uri="nxbre://binder">CalculateTotalWeight</Ind>** </Atom> </head> (...)	Compute the new value of the fact "Chocolate_Box_Weight" using a formula. Here, a binder is used to define and evaluate the formula referred as "CalculateTotalWeight".

<Implies>
 <oid>
  <Ind>label:update total weight;action:modify</Ind>
 </oid>
 <head>
  <Atom>
   <op>
    <Rel>Chocolate_Box_Weight</Rel>
   </op>
   <Var>Box</Var>
   <Ind uri="nxbre://expression">
    expr:{predicate:1}+{var:Quantity}*{var:Weight}
   </Ind>
  </Atom>
 </head>
 (...)
	Same, but in this case no binder is used and the C# expression is directly stored in the rule.
	
	See chapter 3.5 for more information on the syntax.

3.3.7. Salience and Weight

The engine estimates the salience of implications only for the ones implied in pre-condition hierarchies in order to prioritize correctly if they have the same priority level. Then engine combines the priority and the salience in order to calculate the overall implication weight:

```
Weight = (1 + Priority) * 100 + Salience
```

This weight is used for prioritizing the implication at process time.

3.4. Data Binding Strategies

By design, the Inference Engine does not offer reflection features to directly access business objects. The main reason for this is that the rule files should not be cluttered with programmatic concepts like object introspection but should only work on facts and their underlying predicates. This should allow a business oriented analyst to focus on the rules and, afterwards, to have a programmer to work on the binding of these rules (more precisely the facts they rely on) with the business objects.

NxBRE offers three different strategies for accomplishing this binding.

3.4.1. Basic

The Inference Engine offers the possibility to directly assert facts in the working memory. This is the simplest and fastest way of binding business objects facts. It is also the less versatile strategy because re-compilation is always required.

3.4.2. Rulebase Adapters

The Inference Engine uses Rulebase adapters (i.e. classes implementing `org.nxbre.ie.adapters.IRuleBaseAdapter`) for loading rule bases and fact bases. Only one rule base can be loaded, but when this is done many fact bases can then be loaded.

By writing custom Rulebase adapters, like ones for fetching facts from a RDBMS or a web service, the user can provide facts to **NxBRE** with a different approach than by directly asserting facts in the engine.

The RuleML adapters are just a particular type of adapter: one is free not to use RuleML at all for storing his rule and fact bases.

3.4.3. Business Object Binders

The interface `org.nxbre.ie.adapters.IBinder` defines a more advanced concept for binding business objects. Classes implementing this interface will be called by the engine at specific moments in order to perform specific operations (the classical inversion of control concept) like: assertion of facts, action on new facts, post-analysis of facts.

The Business Object Binders work in two distinct modes:

- the Control mode, where the binder orchestrates the facts assertion and engine processing.
- the BeforeAfter mode, where the engine calls the binder at specific moments for performing specific operations.

Please refer to chapter 3.7 for a detailed view of execution time and to the API Documention for the detail of the IBinder contract (see page 46).

Binders not only perform assertion of facts based on business objects but also evaluate function predicates. For example in the implication shown page 20, the binder could recognize the individual predicate “min 5000 eur” as a function predicate (based on a regular expression for example), and then be called whenever a fact should be evaluated against the atom containing the function.

NB. The Binder is not used for evaluating built-in operators, like **`NxBRE:LessThan(x)`**. In that case, the individual predicate is directly evaluated by calling the helper object.

NxBRE performs these evaluations at assertion time, therefore, while the processing will always be very fast, the assertion of facts from the business objects will be significantly slower if many functions have to be evaluated.

Implementing IBinder in a class of the project provides the fastest binding method but also the less versatile one, as modification would require rebuilding the whole project. **NxBRE** comes with two implementations of IBinder, one that uses the Flow Engine (see chapter 2) for controlling the binding from XML files and one that performs on-the-fly compilation of class files implementing IBinder.

If the Flow Engine binding is two slow or complex, a good approach consists in designing and testing the binder in the development environment, then removing it from the project and storing it as a file alongside the rule files, then using on-the-fly compilation.

This approach also offers the advantage of easily debugging the binder: keep it in your source code until it is stable enough, then externalize it.

To summarize, the Binder is responsible for:

- asserting facts based on business objects and data sources,
- determining if an individual predicate must be interpreted as a function and then is called-back to evaluate these functions when needed by the engine,
- evaluating custom function-based atom relations,
- asserting / retracting facts based on the results of a inference process (in this sense, by leveraging the After part of the Before/After binder, implementers can perform data drill down conducted by rules).

Think of the Binder as the code-behind your Rule Base. It is essential to get familiar with this concept to truly leverage the Inference Engine of **NxBRE**.

3.5. Expression Language Support

NxBRE supports C# based expressions that can be used in four places:

1. in the relation definition of a function based atom relations (see chapter 3.3.5),
2. in the individual comparison functions (see chapter 3.1.5),
3. in formulas for computing individual predicate values in the deduction part of an implication (see chapter 3.3.6),

Expressions are evaluated to return a value: the type of the returned value depends of the evaluation context.

Return Type	Evaluation Context
boolean	1 and 2 in above list
object	3 and 4 in above list

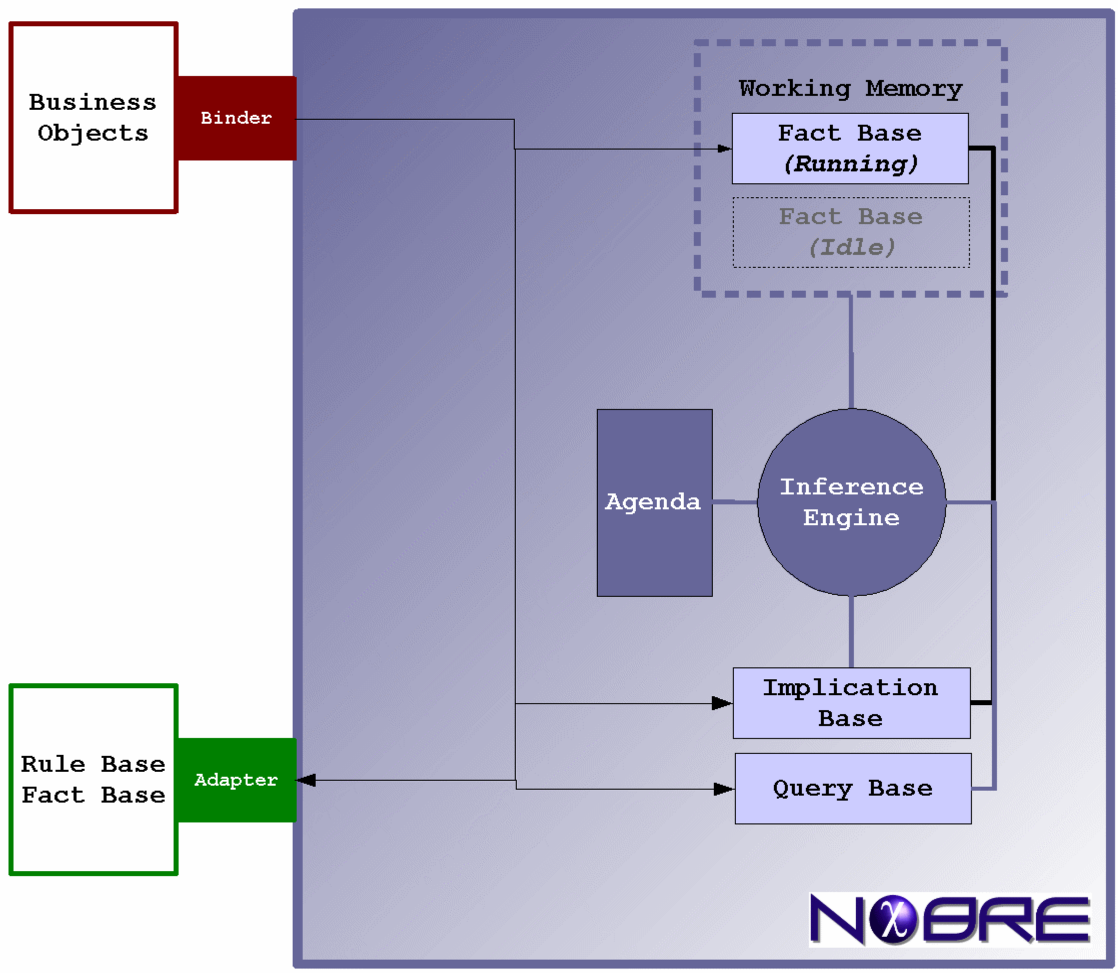
This powerful feature can be interesting for projects where having technical concepts in the rule file is acceptable. In this case, using a binder is not required for performing the different evaluations detailed here above ; but it can still be very useful for dynamic fact assertions and events processing.

C# expressions are compiled at first use: the predicates types are then frozen. This means that in a certain predicate, you must use the same type (or at least castable types) throughout your application. Failing to do so will result in casting errors at run-time.

The C# expression supports three types of variable place-holders:

Placeholder	Description
{var:XYZ}	Contains the value of a variable predicate named XYZ, in the body part of an implication or query.
{ind}	Contains the value of the matched predicate. Used only for individual match evaluation (see chapter 3.1.5).
{predicate:N}	Contains the value of the Nth predicate (0 based count) of a matching atom. Used only for deduction atom of a modifying implication (see chapter 3.3.6).

3.6. Engine



3.6.1. Working Memory

The Working Memory is composed of Running Fact Base and potentially an Idle Fact Base, the existence of the latter being based on the operation mode, which could be Global, Isolated or IsolatedEmpty.

3.6.1.a. Fact Base

The Fact Base main index is a long hashcode calculated for each fact in order to store and retrieve them very efficiently. Each fact has a unique hashcode based on its type and a combination of the hashcodes of each predicate and their position. Consequently, the Fact Base can not contain two identical facts: trying to assert an already existing fact will leave the base unchanged.

The hashcode is a long integer (Int64) and not a regular integer (Int32) in order to reduce the risk of similar hashcodes when inferring on large fact bases (hundreds of thousands). Even if the hashcode of each predicate is re-hashed using MD5 in order to ensure that two very similar predicates produce two very different hashcodes, it is recommended not to rely on the default implementation of GetHashCode()[4] but to override it with a significant one (like the PK of the predicate).

The Fact Base also stores extra hashcodes of each fact representing all the possible ways of querying them from the matching atoms in the loaded implications. The following table shows all the matching atoms for which hashcodes will be calculated. The atom representation uses question marks for variables.

Atom Type	Atom/Fact[5] for which Hashcode will be stored
Core Fact	`Spending{John Q. Doe, 1000 EUR}`

Core Derived	`Spending{?, 1000 EUR}`
	`Spending{1000 EUR, ?}`
	`Spending{?, ?}`

Matched from Implications	`Spending{?, min(5000 EUR)}` `Spending{?, min(0 EUR)}`

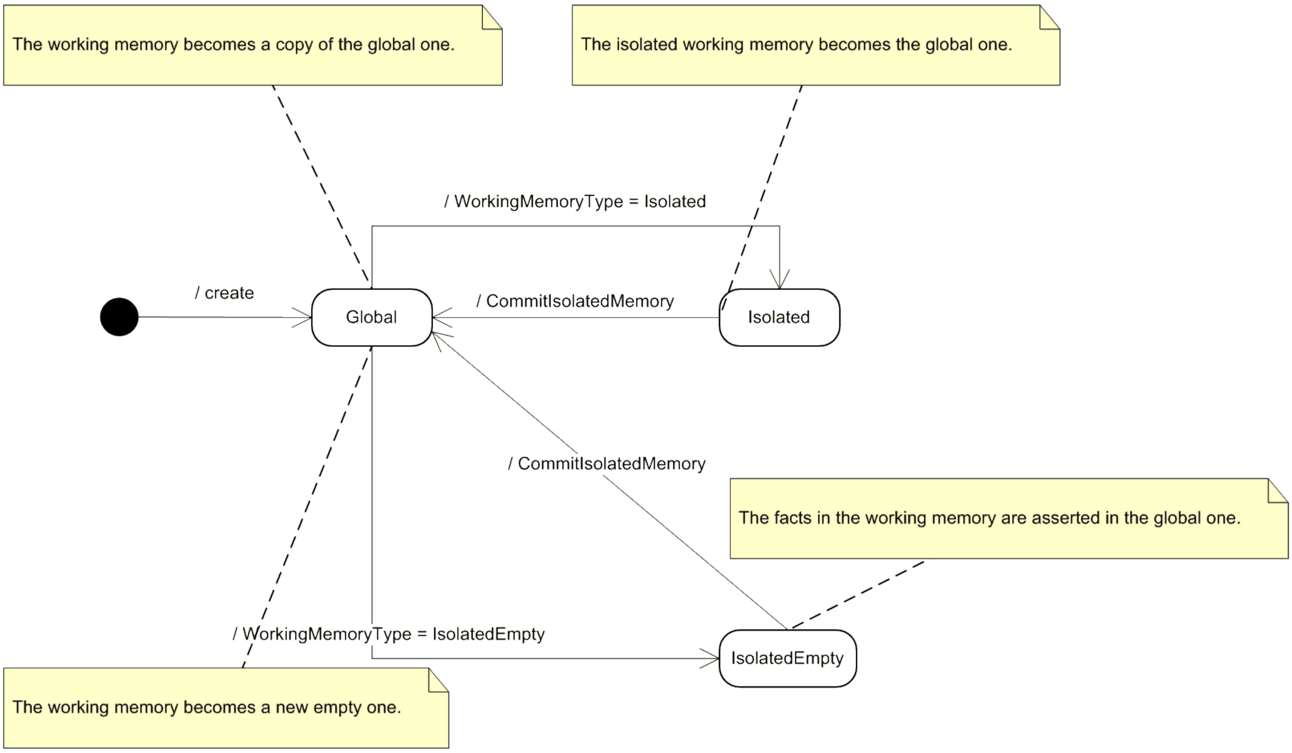
For critical systems, when loading facts in the engine, it is recommended to check that the asserted facts where all accepted by the Fact Base to ensure that no collision occurred (meaning that the predicate hashcodes are of good quality).

3.6.1.b. Global, Isolated and IsolatedEmpty Modes

When the Working Memory is in Global mode, all facts exists in a single Fact Base. This is typically well adapted for a knowledge base that gets enriched as the engine infers and that is persisted regularly.

The Isolated mode is created by cloning the Global mode Fact Base then asserting all new facts in this clone. This clone is unique and transient in the sense that it is discarded when the mode is changed. There is the possibility of "committing" the Isolated Memory, making it the new Global one (for example if a process produced valid or expected results). The Isolated mode is typically used when the knowledge based is constant (facts and implications initially loaded from files in the Global memory) and the asserted and deducted facts concern business objects undergoing an evaluation process.

4 See also: ms-help://MS.NETFrameworkSDKv1.1/cpref/html/frlrfSystemObjectClassGetHashCodeTopic.htm
5 In Human Readable Format, see chapter 3.10.



The IsolatedEmpty mode is created by instantiating a new empty Fact Base and using it as the running one. This new Fact Base is also discarded when the mode changes ; but in this case the commit feature performs an individual assertion of all the facts of the running memory in the idle one, and establish the idle one as the new running one, while switching the mode back to Global.

NB. Using Isolated memory does not allow multi-threading process as it is not possible to fork different Isolated memories for different threads from a single Global memory.

3.6.2. Agenda

The Agenda is the object responsible for scheduling the implications for the engine for evaluation. The order of execution is based on the implication weight (see chapter 3.3.7).

The Agenda decides what implications must be scheduled by analyzing the results of the previous iteration in the process cycle (see chapter 3.7). Based on what implications were positive, the facts they potentially asserted and the other implications they potentially preconditioned, the Agenda schedules just the implications that are worth evaluating for the next iteration.

3.6.3. Implication Base

The Implication Base contains all the loaded implications. By design it is not possible to programmatically alter this base. Instead it is recommended to persist the rule base, amend it outside of **NxBRE** and reload it, **NxBRE** being only the execution environment.

3.6.4. Query Base

The Query Base contains all the loaded queries. Like for implications it is not possible to programmatically add new queries in the base ; but it is possible to run new queries on the current Working Memory.

3.7. Execution

The process cycle of the Inference Engine (`org.nxbre.ie.IEImpl`) is very simple: basically it infers as long as new facts are deducted. If during an iteration all the deducted facts where already present in the Fact Base, then the process stops. There is also a limit[6] on the maximum number of iterations allowed in a process cycle: if this limit is reached, the engine stops and throws an exception.

Hereafter a more detailed explanation of the process cycle:

1. If there is a Business Object Binder in Control mode, transfer control to it.
2. If there is a Business Object Binder in BeforeAfter mode, call its BeforeProcess method.
3. If the maximum number of iteration is reached, throw an Exception.
4. Ask the Agenda to schedule the necessary implications.
5. Evaluate all the scheduled implications.
6. If new facts where asserted during point 5, perform a new iteration → 3.
7. If there is a Business Object Binder in BeforeAfter mode, call its AfterProcess method.
8. If new facts where asserted or retracted during point 7, perform a new iteration → 3.

6 The default limit is set to 1000. This parameter (IEImpl.IterationLimit) can be changed.

3.8. Threading Model

Since version 2.4, the Inference Engine offers three threading models:

- **Single**, for mono-threaded applications,
- **Multi**, for multi threaded applications,
- **Multi Hot Swap**, for multi threaded applications with the need for hot swapping rule base and/or binder at run-time without suspending the execution of the hosting application.

3.8.1. Implementation sample

To leverage this feature, it is compulsory to use IsolatedMemory as each thread will be assigned an instance of its own. The following code demonstrates a sample implementation, in the case a binder is used:

```
(Initialization)
01   IInferenceEngine ie = new IEImpl(binder , ThreadingModelTypes.Multi);
02   ie.LoadRuleBase(adapter);
(Usage)
03   ie.NewWorkingMemory(WorkingMemoryTypes.Isolated);
04   ie.Process(businessObjects);
```

The initialization phase can happen in a singleton and can even be a static instance.

The important thing to bear in mind is that you should have one instance per rule file you want to run in parallel: so if your application can potentially process simultaneously 5 different rule bases, you should have 5 engine instances somewhere (a Dictionary-based registry would be indicated in that case).

Line 03 shows how to instantiate a non-empty isolated memory, which will be specific to the current thread (an empty isolated memory would also work).

3.8.2. Hot swapping support

This feature allows any thread to perform a rule base and/or binder reload at any time, without interrupting the application: one can imagine monitoring file system events and trigger a reload operation if a file has changed.

The following sample shows how to leverage this ability.

```
(Initialization)
01   IInferenceEngine ie = new IEImpl(ThreadingModelTypes.MultiHotSwap);
(Initialization & Hot Swap)
02   ie.LoadRuleBase(adapter, binder);
(Usage)
03   ie.NewWorkingMemory(WorkingMemoryTypes.Isolated);
04   ie.Process(businessObjects);
05   ie.DisposeIsolatedMemory();
```

Line 02 shows how to reload the rule base and the binder at the same time. It is possible to reload the rule file only, but it is not possible to reload the binder only.

Line 05 shows how to specifically free this isolated memory. Note that this syntax is strictly equivalent to:

```
ie.NewWorkingMemory(WorkingMemoryTypes.Global)
```

but it is simply much clearer. Freeing the working memory is not compulsory but is a good practice as it releases internal locks and could then allow a reload operation, which is exclusive, to happen.

When a reload is initiated the engine will suspend all new isolated memory requests until it can perform the rule base / binder swap ; then it would release the locks and allow the other threads to work again and acquire isolated memories.

These lock operations are conditioned by a time-out that is statically set on the engine:

IEImpl.LockTimeOut = <time out time in milliseconds>[7]

This is to prevent a deadlock situation of the engine. If a single processing on your rule base takes more than this time-out, there is the risk that the hot swapping will never happen and exceptions will be thrown. Tune this value according to your applications performances.

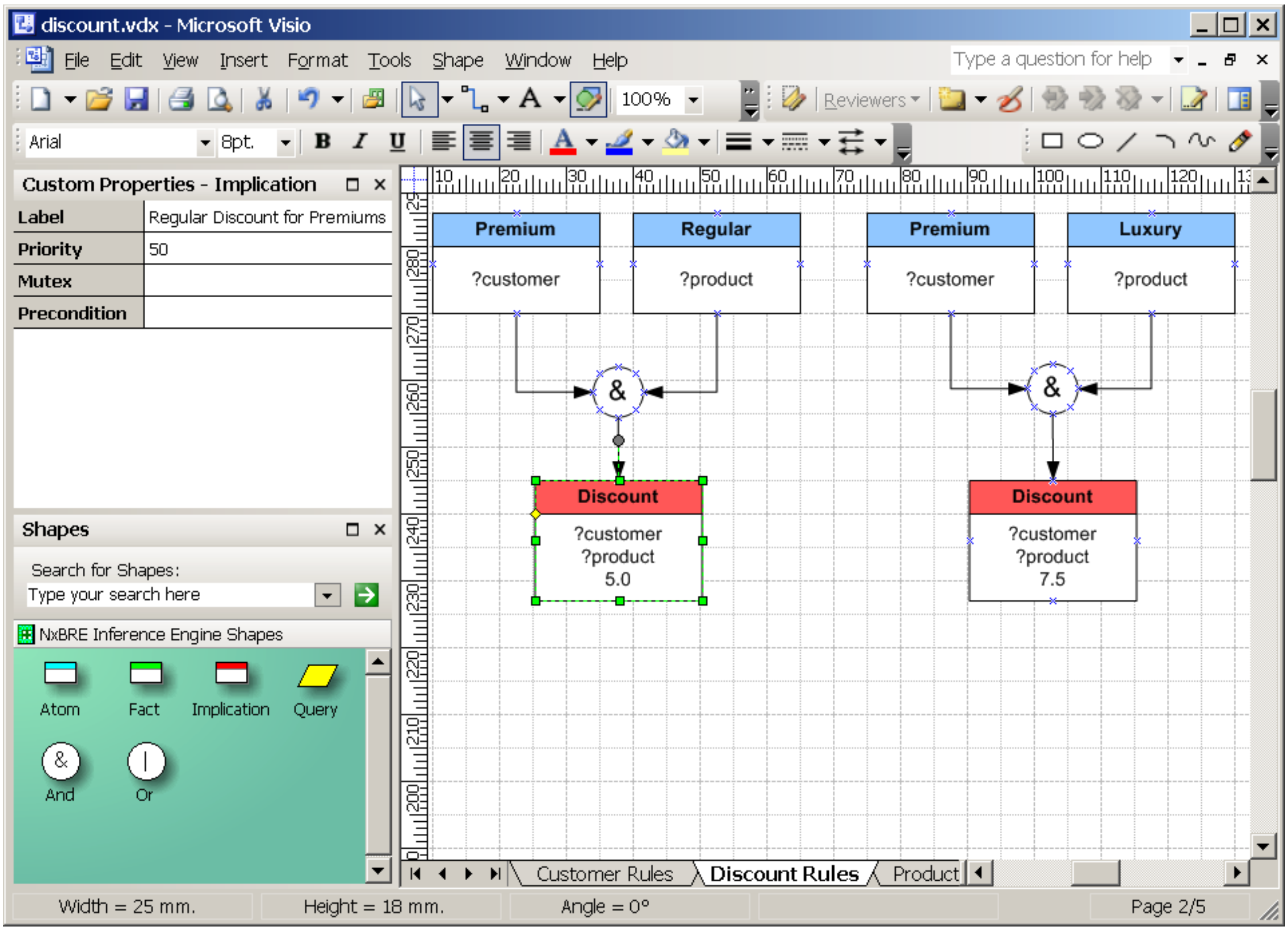
7 The default is 15000 milliseconds.

3.9. Microsoft Visio 2003 Adapter

In order to provide a convenient environment for creating rule bases, we have decided to leverage both Microsoft Visio 2003 excellent user interface and its XML format called DatadiagramML.

This choice of a commercial product for an open source project can look odd, but we wanted to avoid an editor we would have made from scratch, exposing the users to potential bugs and poor ergonomics.

With the definition of a **small set of specific shapes** (available in a stencil provided with this distribution: `NxBRE-IE.vsx`), and by using **standard dynamic connectors** to connect these shapes together, it becomes possible to easily develop rule bases.



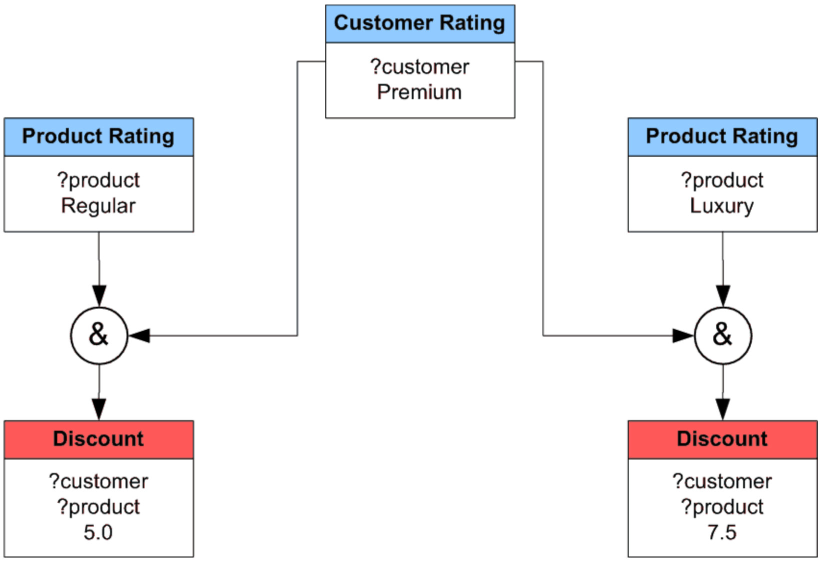
This edition model is extremely simple, but not fool proof: currently, there is no enforcement of what shapes you use and connect together, so it is possible to create meaningless rule bases. In that case it is most likely that the **NxBRE** adapter will reject the rule base ; but be aware that this is not guaranteed.

The immediate advantage is to allow developers to model in a unique environment (Visio) all their systems in UML, connecting use cases and other UML models to business rules expressed with **NXBRE**'s stencil.

Ultimately, code generation tools should allow a complete generation of business objects, rule base files and the scaffolding code (engine instantiation and fact binding).

There are different direct bonuses for designing rule bases in Visio:

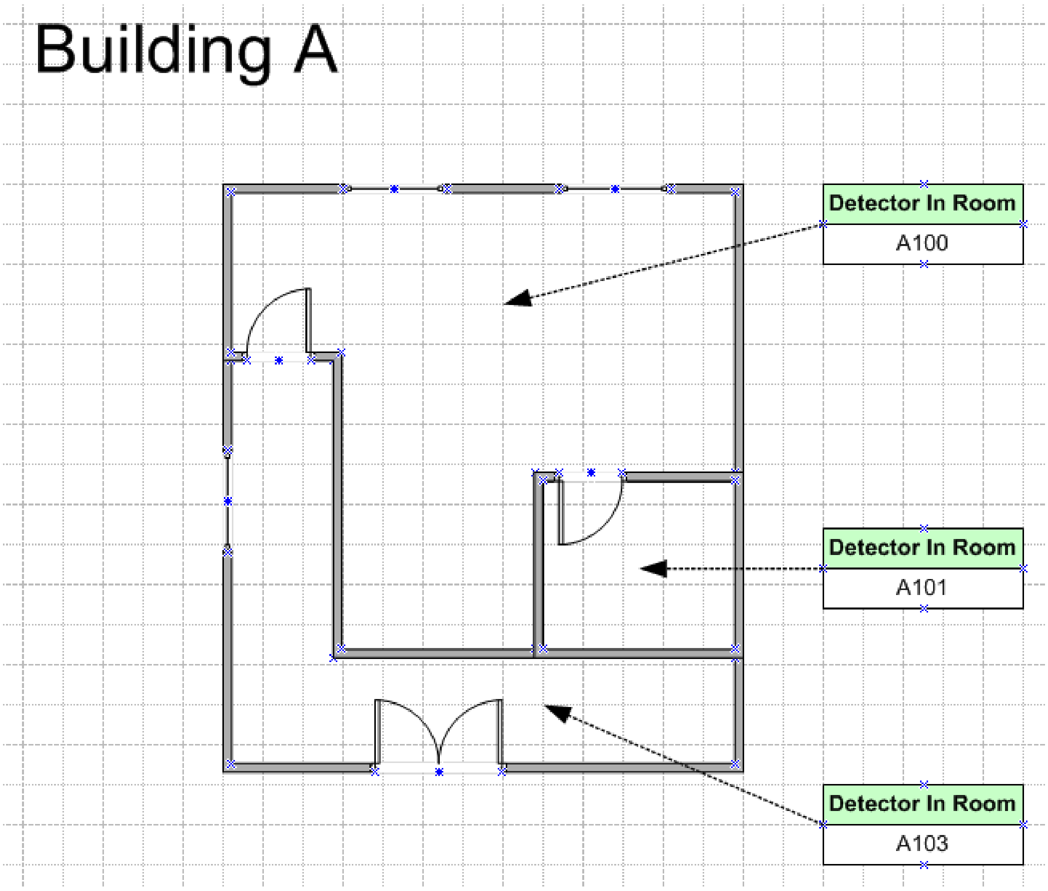
- **Multi pages**: you can organize your implications, facts and queries on as many pages as you want, allowing logical grouping of entities. On top of that, the **NxBRE** adapter is able to load only a selection of these pages, providing a sub-grouping that is absent from RuleML.



- **Atom syndication**: as shown above, it is possible to syndicate atoms that are used by different implications (or queries), limiting potential mistakes and increasing readability by reducing duplicated information.
- **Easy operators**: when transforming DataDiagramML to RuleML, the XSL-T takes care of transforming basic operators to their **NxBRE** counterparts, as shown in the following table:

<table>
<tr><th>Visio Symbol</th><th>NxBRE Operator</th></tr>
<tr><td>>=</td><td>GreaterThanEqualTo</td></tr>
<tr><td><=</td><td>LessThanEqualTo</td></tr>
<tr><td><></td><td rowspan="2">NotEquals</td></tr>
<tr><td>!=</td></tr>
<tr><td>==</td><td rowspan="2">Equals</td></tr>
<tr><td>=</td></tr>
<tr><td><</td><td>LessThan</td></tr>
<tr><td>></td><td>GreaterThan</td></tr>
</table>

- **Artifact mix**: as shown below, it is possible to mix design artifacts coming from different stencils in a rule base, the only constraint being to avoid dynamic connections between **NxBRE**'s artifacts and other ones.



TIP: The rule base label is taken from the Title attribute of the File Properties.

3.10. Human Readable Format (experimental)

The user can find in the resource folder a file named **ruleml2hrf.xsl** that can be useful for generating human readable rule bases from RuleML 0.86 NafDatalog files.

Human Readable Samples

```
premium{?customer}
& regular{?product}
-> discount{?customer, ?product, 5.0 percent};

premium{?customer}
& luxury{?product}
-> discount{?customer, ?product, 7.5 percent};

spending{?customer, min 5000 euro, previous year}
-> premium{?customer};

luxury{Porsche};

regular{Honda};

spending{Peter Miller, min 5000 euro, previous year};

discount{?customer, ?product, ?amount};

[Safe Room List]
Room In Zone{?Room Number, ?Zone Number}
& Firemen In Room{?Room Number}
| ! Fire Alarm In Room{?Room Number}
& ! Alarm Fault In Room{?Event Type, ?Room Number};

[Test A and B or C]
id{?account, NxBRE:LessThan(500)}
& balance{?account, NxBRE:GreaterThanEqualTo(100)}
| balance{?account, NxBRE:LessThanEqualTo(50)}
-> testAandBorC{?account};
```

An adapter able to read and write rule bases in this format as been developed by Andre Weber using Coco/R. This adapter has been improved by Ron Evans to support the latest language constructs of RuleML 0.86 NafDatalog.

Currently, the adapter does not support complex nesting of AND/OR and only the US-ASCII encoding is supported ; but it is already able to be used for simple rules (as shown above).

4. API Documentation

The API Documentation is available as a CHM file that should be bundled with the current document.

NB. The on-line HTML API Documentation that was previously available has been discontinued, as it was not very much useful and painful to maintain.

5. Support

For comments or questions use the SourceForge forums or write to: contact@nxbre.org

You can also enter bugs and feature requests on SourceForge.

Additional support can be found at the NxBRE Wiki Knowledge Base.