

.NET Business Rules Engine

<http://nxbre.org>



David Dossot, Project Lead

Version 3.0.0
September 8, 2006



Table of Contents

1. Introduction.....	4
1.1. What is NxBRE?.....	4
1.2. What is this documentation?.....	5
1.3. Release Notes.....	5
1.4. Content of the Rulefiles folder.....	5
1.5. How to choose between the Flow and the Inference Engine ?.....	5
2. The Flow Engine.....	6
2.1. Introduction.....	6
2.2. The IFlowEngine interface.....	7
2.3. Rule Interpreter Implementation: NxBRE.FlowEngine.BREImpl.....	8
2.3.1. Execution.....	9
2.3.2. Engine.....	9
2.3.3. Helper Objects.....	9
2.3.4. Rule Drivers.....	10
2.3.5. Factories.....	11
2.3.6. Threading Model.....	11
2.4. Rule Files Formats.....	12
2.4.1. Language constructs of the extended syntax.....	12
2.4.1.a. Data assertion.....	12
2.4.1.b. Reflection and delegation calls.....	13
2.4.1.c. Increments.....	13
2.4.1.d. Operators and conditions.....	13
2.4.1.e. Logic blocks.....	14
2.4.1.f. Sets.....	14
2.4.1.g. Exceptions and logging.....	15
2.4.2. Native and extended syntax comparison.....	15
2.4.3. Pseudo-code Rendering.....	17
2.4.4. Rules Format Comparison.....	17
3. The Inference Engine: NxBRE.InferenceEngine.....	21
3.1. RuleML Naf Datalog Concepts.....	21
3.1.1. Atoms.....	21
3.1.2. Logical Operators.....	22
3.1.3. Facts.....	22
3.1.4. Queries.....	22
3.1.5. Implications.....	23
3.1.6. Slots.....	24
3.1.7. Multi-syntax.....	24
3.1.8. Typed data.....	25
3.1.9. Deterministic resolution.....	26
3.1.10. Equivalence.....	27
3.1.11. Integrity protection.....	28
3.1.12. Migrating from a previous version.....	29
3.2. Advanced Concepts.....	30
3.2.1. Priority.....	30



3.2.2. Mutual Exclusion.....	30
3.2.3. Pre-Condition.....	31
3.2.4. Implication Action.....	32
3.2.5. Function Based Atom Relations.....	33
3.2.6. Formula.....	34
3.2.7. Saliency and Weight.....	34
3.3. Data Binding Strategies.....	35
3.3.1. Basic.....	35
3.3.2. Rulebase Adapters.....	35
3.3.3. Business Object Binders.....	35
3.4. Expression Language Support.....	37
3.5. Engine.....	38
3.5.1. Working Memory.....	38
3.5.1.a. Fact Base.....	38
3.5.1.b. Global, Isolated and IsolatedEmpty Modes.....	39
3.5.2. Agenda.....	40
3.5.3. Implication Base.....	40
3.5.4. Query Base.....	40
3.6. Execution.....	41
3.7. Threading Model.....	42
3.7.1. Implementation sample.....	42
3.7.2. Hot swapping support.....	43
3.8. Microsoft Visio 2003 Adapter.....	44
3.9. Human Readable Format (experimental).....	47
3.10. Registry.....	48
3.10.1. Concepts.....	48
3.10.2. File Registry.....	48
3.11. Performance tuning.....	49
3.11.1. Have strongly identified facts.....	49
3.11.2. Use small facts.....	49
3.11.3. Use typed data and storage.....	49
3.11.4. Order atoms in And blocks.....	50
4. Configuration.....	51
5. Logging.....	52
6. API Documentation.....	53
7. Support.....	54
8. Other engines.....	55
8.1. Open source engines.....	55
8.1.1. Drools DotNet.....	55
8.1.2. Simple Rule Engine (SDSRE).....	55
8.2. Commercial engines.....	55

Disclaimer: Throughout this guide, all cited trademarks belong to their respective owners.



1. Introduction

1.1. What is NxBRE?

NxBRE is the first¹ 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 started as a port of [JxBRE](#) v1.7.1 (from [Sloan Seaman](#)).
- 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 embed it in commercial solutions.

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 Chronologically speaking.



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`², 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** by one particular Flow Engine, the **Rule Interpreter** : the XML rule file is parsed each time a process is launched and each rule is interpreted when they are read,

² Though `IBRERule` would have been a better name, the original `JxBRE` name has been kept (prefixed with `I`).



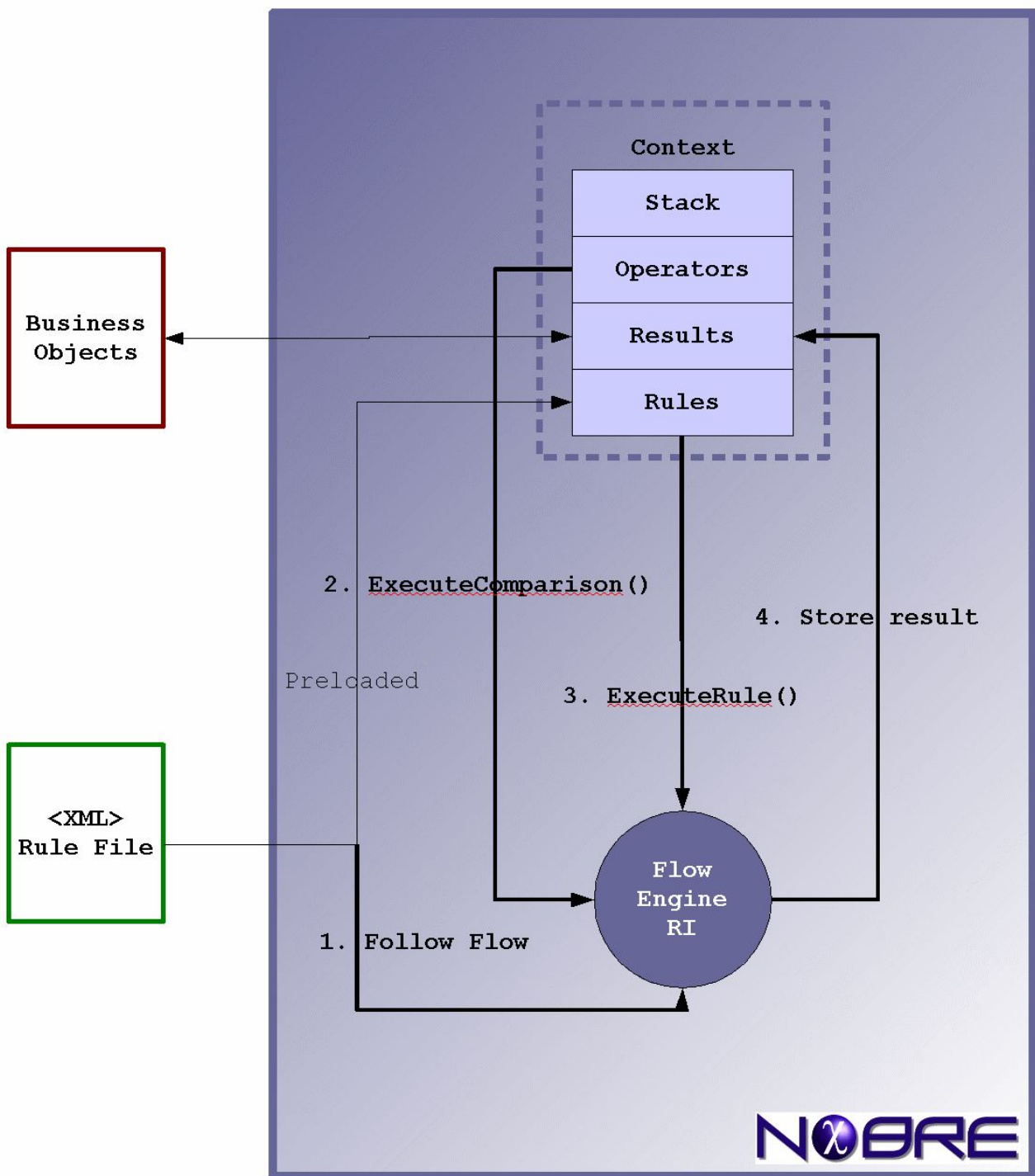
2.2. The IFlowEngine interface

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.

Method, Property or Event	Package.Interface	Description
<code>DispatchRuleResult</code>	<code>NxBRE.FlowEngine.IBREDispatcher</code>	Called when a result is added to the context.
<code>Init</code>	<code>NxBRE.FlowEngine.IFlowEngine</code>	Defines the different ways to initialize the engine.
<code>RuleContext</code>		The engine's context.
<code>XmlDocumentRules</code>		The loaded XML rules.
<code>Process</code>		Start the engine and process all accessible rules.
<code>Process(ruleSetId)</code>		Start the engine and process all rules in a set.
<code>Stop</code>		Stop the engine abruptly.
<code>Reset</code>		Place the engine in a state where previous results are cleared, ready to process again.
<code>Clone</code>	<code>System.ICloneable</code>	Returns a clone of the engine, ready to process. The cloning depth should not extend to the objects in the context.



2.3. Rule Interpreter Implementation: *NxBRE.FlowEngine.BREImpl*





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 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: <code>NxBRE.FlowEngine.Rules</code>	Description
<code>Decrement*</code>	Integer that decrements.
<code>Exception</code>	Raises an exception.
<code>False</code>	Constant boolean false.
<code>FatalException</code>	Raises a fatal exception.
<code>Increment*</code>	Integer that increments.

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



Class name in: <code>NxBRE.FlowEngine.Rules</code>	Description
<code>IncrementInit*</code>	Incrementor or Decrementor reset.
<code>ObjectLookup</code>	Reflection call on a class or an object.
<code>True</code>	Constant boolean true.
<code>Value</code>	Instantiation of any type.

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

Class name in: <code>NxBRE.FlowEngine.Rules</code>	Description
<code>Equals</code>	<code>==</code>
<code>GreaterThan</code>	<code>></code>
<code>GreaterThanEqualTo</code>	<code>>=</code>
<code>InstanceOf</code>	<code>InstanceOf SubtypeOf</code>
<code>LessThan</code>	<code><</code>
<code>LessThanEqualTo</code>	<code><=</code>
<code>NotEquals</code>	<code>!=</code>

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: <code>NxBRE.FlowEngine.IO</code>	Description
<code>BusinessRulesFileDriver</code>	Loads up rules in the <i>native</i> format, i.e. defined by <code>businessRules.xsd</code> (see chapter 2.4).
<code>XSLTRulesFileDriver</code>	Loads rules in any custom format and transform it to the native format by performing an XSLT.
<code>XBusinessRulesFileDriver</code>	Specialization of the previous driver that transforms rules defined by <code>xBusinessRules.dtd</code> (see chapter).

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



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.

They are presented in the following table.

Class name in: <code>NxBRE.FlowEngine.Factories</code>	Description
<code>BREFactory</code>	Instantiate an engine with a specific driver and optional event handlers.
<code>BREFactoryConsole</code>	Specialization of the previous factory that registers handlers that write their events to the console.
<code>BRECloneFactory</code>	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

The engine supports two rule languages, defined by two different schema:

- `businessRules.xsd`: the native syntax, coming from **JxBRE**,
- `xBusinessRules.xsd`: the extended syntax, which can easily be XSL-Transformed into the native syntax.

The new rule format, which is the format of choice for the Flow Engine, 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 semantics leading to a confusing syntax.

2.4.1. Language constructs of the extended syntax

Tip: Use a decent XML editor for writing rule base. It should provide XML element insertion assistance.

NB. All examples come from the provided test file: `test.xbre`

2.4.1.a. Data assertion

xBusinessRules sample	Description
<code><Assert id="5i" type="Integer" value="5"/></code>	Create a variable in the memory context named "5i", of type "Integer" and value "5". System types are: Exception, Boolean, Byte, Short, Integer, Long, Single, Double, Decimal, Date, DateTime, Time and String.
<code><Assert id="TestObject2" type="NxBRE.Test.TestObject"> <Argument valueId="STORED_TRUE"/> <Argument value="99" type="Integer"/> <Argument value="world"/> </Assert></code>	A fully qualified class name, with optionally the assembly name after a coma, is also acceptable. Constructor arguments can be passed, if needed.
<code><Boolean id="TRUE" value="true"/> <Byte id="8b" value="8"/> <Date id="xmas2003" value="2003-12-25"/> <DateTime id="manOnMoon" value="1969-07-21T02:56:00"/> <Time id="wakeUpCall" value="07:15:30"/> <Decimal id="3.14m" value="3.14"/> <Double id="3.14d" value="3.14"/> <Integer id="ZERO" value="0"/> <Integer id="10i" value="10"/> <Short id="16s" value="16"/> <Single id="3.14" value="3.14"/> <String id="hello" value="world"/></code>	For system types, an element can also be used to create the variable. This is the best construct for system types.
<code><False id="STORED_FALSE"/> <True id="STORED_TRUE"/></code>	Booleans even exist in a shorter form!



2.4.1.b. Reflection and delegation calls

xBusinessRules sample	Description
<code><ObjectLookup id="TestObject_MyField" objectId="TestObject" member="MyField"/></code>	Reads the value of a member of a particular object (referenced by objectId) and stores its value in the context under the given id.
<code><ObjectLookup id="TestMultiply" type="NxBRE.Util.Maths" member="Multiply"> <Argument value="2" type="Integer" /> <Argument value="5" type="Integer" /> <Argument value="9" type="Integer" /> </ObjectLookup></code>	Performs a call to a static member of an helper class, providing arguments for the call, and storing the result in the context under the given id.
<code><ObjectLookup objectId="TestObject" member="MyField"> <Argument valueId="STORED_FALSE"/> </ObjectLookup></code>	Calls a particular member of a particular object (referenced by objectId), providing arguments for the call. The engine's context is not modified.
<code><Evaluate id="GlobalCounter"/></code>	Calls the .NET delegate that have been bound under id in the engine context prior to processing the rule base.

2.4.1.c. Increments

xBusinessRules sample	Description
<code><Increment id="INC_X" step="1"/></code>	Adds 1 to the increment stored in the context under the given id.
<code><Increment id="INC_X" value="6"/></code>	Set the increment value to 6.
<code><Increment id="INC_Y" valueId="ZERO"/></code>	Set the increment value to the content of the context under the given valueId.

2.4.1.d. Operators and conditions

xBusinessRules sample	Description
<code><Equals leftId="TRUE" rightId="VALUE1"/></code>	Checks if the content of the context under leftId is equal to the content of the context under rightId.
<code><Equals leftId="T1" rightId="V1"> <True id="T1"/> <Assert id="V1" type="Boolean" valueId="STORED_TRUE"/> </Equals></code>	Values can be asserted in the context at the operator level: in this case, the values for leftId and rightId are first asserted as described inside the Equals operator element before being compared.
<code><And> <IsTrue valueId="STORED_TRUE"/> <IsFalse valueId="STORED_FALSE"/> </And></code>	A simple logical AND block.
<code><Or> <NotEquals leftId="TRUE" rightId="VALUE1"/> <Equals leftId="TRUE" rightId="VALUE1"/> </Or></code>	A simple logical OR block.



xBusinessRules sample	Description
<pre> <And> <IsAsserted valueId="testAssert"/> <Not> <IsAsserted valueId="turnip"/> </Not> </And> </pre>	<p>Conditional blocks can be nested as need be.</p>

2.4.1.e. Logic blocks

xBusinessRules sample	Description
<pre> <Logic> <If> <And> <Equals leftId="STORED_TRUE" rightId="VALUE1"/> </And> <Do> <ThrowFatalException value="LT3"/> </Do> </If> <Else> <True id="LT3"/> </Else> </Logic> </pre>	<p>A logic block should contain a main condition (IF) and can contain an alternative (ELSE).</p> <p>The main condition first contains a conditional top element (AND, OR, NOT) then the action element (DO).</p> <p>An action element can contain any valid operation (including another logic block).</p> <p>The alternative contains the operation(s) to execute.</p>
<pre> <While> <And> <LessThan leftId="INC_X" rightId="10i"> <Increment id="INC_X" step="1"/> </LessThan> </And> <Do> <Evaluate id="WhileCounter"/> </Do> </While> </pre>	<p>A while block first contains a conditional top element (AND, OR, NOT) then the action element.</p>
<pre> <ForEach id="ForEachParser" valueId="GetEnumerable"> <Evaluate id="ForEachTester"/> </ForEach> </pre>	<p>Enumerate the object stored in the context under valueId and place its current value under id.</p> <p>Any operation is valid in a ForEach block.</p>

2.4.1.f. Sets

xBusinessRules sample	Description
<pre> <Set id="BROKENSET"> ... </Set> </pre>	<p>Defines a set which is a group of any operations (except Set) identified by an id.</p>
<pre> <InvokeSet id="REFLECTION"/> </pre>	<p>Asks the engine to process the set identified by its id. It is equivalent to call Process(id) on IFlowEngine.</p>
<pre> <InvokeSet valueId="TestObject2"/> </pre>	<p>Asks the engine to process the set identified by the value stored in the context under valueId.</p>



2.4.1.g. *Exceptions and logging*

xBusinessRules sample	Description
<code><ThrowException/></code>	Posts an empty Error event to the FlowEngineRuleBase trace source
<code><ThrowException id="EXCP" value="This is another exception!"/></code>	Alternative syntax that provides a text for the event and stores it in the engine context under the provided id.
<code><ThrowException valueId="hello"/></code>	Alternative syntax that uses the text that is stored in the context under valueId for the event.
<code><ThrowFatalException value="RT1"/></code>	Posts a Critical event with the provided text to the FlowEngineRuleBase trace source.
<code><Log msg="WORKINGSET finished" level="5"/></code>	Posts an event with the provided text to the FlowEngineRuleBase trace source. The level is used as a numeric index in the TraceEventType enumeration. If no matching type is found, Information is used.
<code><Log msgId="hello" level="3"/></code>	Alternative syntax that uses the text that is stored in the context under msgId for the event.

2.4.2. *Native and extended syntax comparison*

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

xBusinessRules Element	businessRules Element	Factory(ies) in NxBRE.FlowEngine.Rules
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 NxBRE.FlowEngine.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 NxBRE.FlowEngine.Rules
True	Rule	NxBRE.FlowEngine.Rules.True
False		NxBRE.FlowEngine.Rules.False

xBusinessRules Element	businessRules Element
Assert	NxBRE.FlowEngine.Rules.Value
Evaluate	NxBRE.FlowEngine.Rules.Value
Modify	NxBRE.FlowEngine.Rules.Value
ObjectLookup	NxBRE.FlowEngine.Rules.ObjectLookup
Increment*	NxBRE.FlowEngine.Rules.Increment
<i>(not kept as it is simply a negative Increment)</i>	NxBRE.FlowEngine.Rules.Decrement
ThrowException	NxBRE.FlowEngine.Rules.Exception
ThrowFatalException	NxBRE.FlowEngine.Rules.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

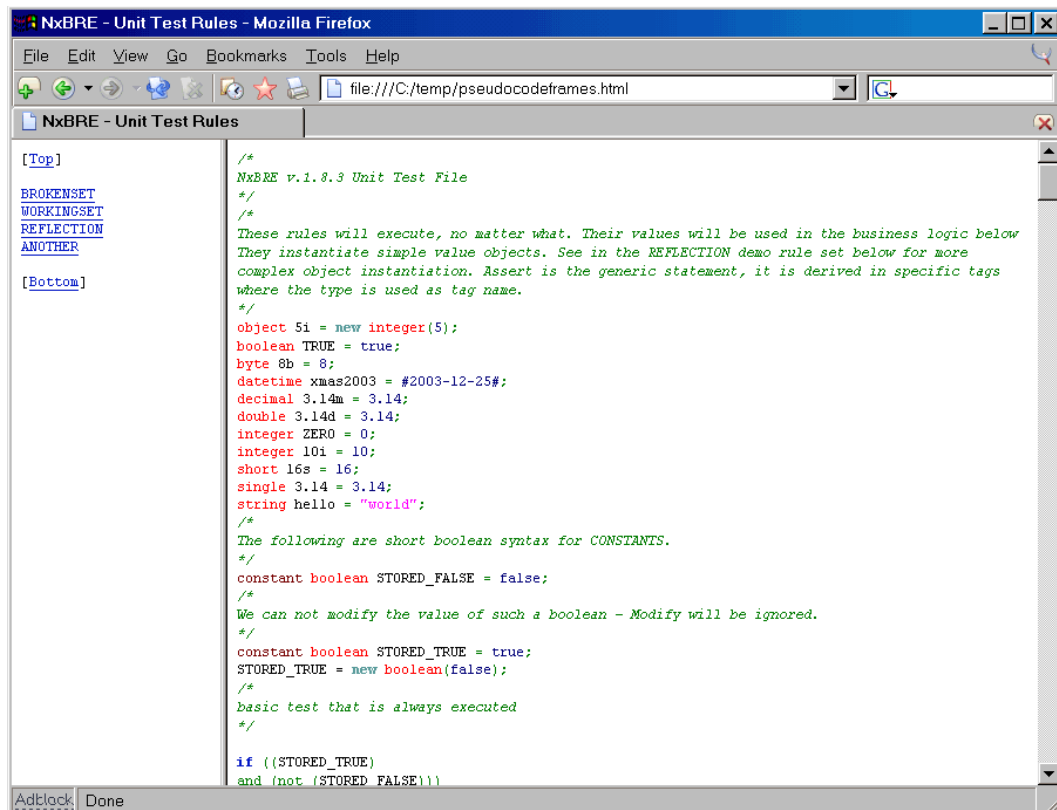
* This helper is stateful therefore not to be used in a multi-threaded approach.



2.4.3. 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.

2.4.4. 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.



```
<BusinessRules xmlns:xsi:noNamespaceSchemaLocation="http://nxbre.org/businessRules.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Rule id="10i" factory="NxBRE.FlowEngine.Rules.Value">
    <Parameter name="Value" value="10"/>
    <Parameter name="Type" value="System.Int32"/>
  </Rule>
  <Rule id="40i" factory="NxBRE.FlowEngine.Rules.Value">
    <Parameter name="Value" value="40"/>
    <Parameter name="Type" value="System.Int32"/>
  </Rule>
  <Rule factory="NxBRE.FlowEngine.Rules.ObjectLookup" id="QuantityOrdered">
    <Parameter name="ObjectId" value="CurrentOrder"/>
    <Parameter name="Member" value="Quantity"/>
  </Rule>
  <Logic>
    <If>
      <Condition type="AND">
        <Compare leftId="ClientRating" operator="NxBRE.FlowEngine.Rules.GreaterThanOrEqualTo" rightId="ClientRatingThreshold">
          <Rule factory="NxBRE.FlowEngine.Rules.ObjectLookup" id="ClientRating">
            <Parameter name="ObjectId" value="CurrentOrder"/>
            <Parameter name="Member" value="ClientRating"/>
          </Rule>
          <Rule id="ClientRatingThreshold" factory="NxBRE.FlowEngine.Rules.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="NxBRE.FlowEngine.Rules.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="NxBRE.FlowEngine.Rules.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="NxBRE.FlowEngine.Rules.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>
```



```

<xBusinessRules xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  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: NxBRE.InferenceEngine

3.1. RuleML Naf Datalog Concepts

The inference engine has been developed using RuleML Naf Datalog³ 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...).

Tip: Use a decent XML editor for writing rule base. It should provide XML element insertion assistance.

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

NxBRE can also recognize function predicates by analyzing the values of individual predicates (see chapter 3.3.3).

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

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

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



RuleML	Natural Language
<pre> <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> </pre>	<p>Give pairs of different children in the same family.</p>

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 3.2.5).

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
<pre> <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> </pre>	<p>A customer is premium if their spending has been min 5000 euro in the previous year.</p>

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.

3.1.6. Slots

Slots are a convenient way of naming an element, which allows an easy retrieval in the application.

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

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
<pre><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></pre>	<p>Using a slot allows a comparison made at a predicate level to produce a named value, which is a very efficient and compact syntax.</p> <p>Named values are directly retrieved with variable predicates in the deduction part of the implication.</p>

3.1.7. 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.

3.1.8. 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
<pre><fact> <_head> <atom> <ind>expr:System.DateTime.Parse("1999/12/31 23:59:59")</ind> </atom> </_head> </fact></pre>
Same fact in RuleML 0.9
<pre><Atom> <Data xsi:type="xs:dateTime">1999-12-31T23:59:59Z</Data> </Atom></pre>

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.



3.1.9. 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	<code><ind>NxBRE:Equals(100)</ind></code>
	<code><Ind uri="nxbre://operator">Equals(100)</Ind></code>
Expression	<code><ind>expr:{ind}.StartsWith("hello")</ind></code>
	<code><Ind uri="nxbre://expression">{ind}.StartsWith("hello")</Ind></code>
Binder Resolved Expression	<code><ind>binder:CalculateTotalWeight</ind></code>
	or: <code><ind>CalculateTotalWeight()</ind></code> for regular expression based recognition.
	<code><Ind uri="nxbre://binder">CalculateTotalWeight</Ind></code>
Function Based Atom Relation	<code><rel>expr:{var:Date} &lt; System.DateTime.Now</rel></code>
	<code><Rel uri="nxbre://expression">{var:Date} &lt; System.DateTime.Now</Rel></code>
Binder Resolved Atom Relation	<code><rel>binder:WithinTolerance</rel></code>
	<code><Rel uri="nxbre://binder">WithinTolerance</Rel></code>



3.1.10. 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
<pre> <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> </pre>	<p>Saying that a person owns a stuff is equivalent to saying that the stuff belongs to the person.</p>

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.



3.1.11. Integrity protection

It is possible to write special queries dedicated to verify the integrity of the rule base, ie 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 *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 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”.



3.1.12. 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:

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



3.2. 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.

3.2.1. 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
<pre><Implies> <oid> <Ind>label:Lower;priority:20</Ind> </oid> </head> (...)</pre>	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.

3.2.2. 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
<pre><Implies> <oid> <Ind>label:polite;mutex:mundane</Ind> </oid> </head> (...)</pre>	Defines an implication labeled "polite" that mutexes an implication labeled "mundane".
<pre><Implies> <oid> <Ind>label:mundane;priority:25</Ind> </oid> </head> (...)</pre>	<p>Defines an implication labeled "mundane" whose priority is 25.</p> <p>NB. the mutex with the implication labeled "polite" could have been defined here.</p>



It is very important to understand that the mutex-lock is maintained for the duration of the process cycle (see chapter 3.6). Therefore the granularity of the asserted business objects (see chapter 3.3.3) 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 3.5.1).

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

3.2.3. 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 3.2.7).

As for mutual exclusion, the same important remark concerning facts granularity applies to pre-condition locking (see chapter 3.2.2).

RuleML	Description
<pre><Implies> <oid> <Ind>label:Rule X;precondition:Rule Z</Ind> </oid> <head> (...)</pre>	<p>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.</p>



3.2.4. 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
<pre><Implies> <oid> <Ind>label:Rule X;action:retract</Ind> </oid> </head> (...)</pre>	<p>Defines an implication labeled "Rule X" whose evaluation will retract facts produced by the inference process.</p>
<pre><Implies> <oid> <Ind>label:Rule Y;action:modify</Ind> </oid> </head> (...)</pre>	<p>Defines an implication labeled "Rule Y" whose evaluation will modify the facts found matching the resolved head part of the implication.</p> <p>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.</p> <p>The facts selected by this query will then be modified, with any formula resolved in the context of each of them.</p> <p>The label of the original facts is preserved.</p>
<pre><Implies> <oid> <Ind>label:Rule Z;action:count</Ind> </oid> </head> (...)</pre>	<p>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.</p> <p>Hence it is always positive, and must be used with care.</p>

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 3.6).



3.2.5. 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
<pre><Atom> <op> <Rel uri="nxbre://operator">GreaterThan()</Rel> </op> (...) </pre>	Evaluates if the first predicate is greater than the second one, using an operator provided by NxBRE.
<pre><Atom> <op> <Rel uri="nxbre://binder">IsInRange</Rel> </op> (...) </pre>	Passes the predicates as arguments to the IsInRange custom function defined in the binder associated with the rulebase.
<pre><Atom> <op> <Rel uri="nxbre://expression">({var:X}) &gt;=5</Rel> </op> (...) </pre>	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.



3.2.6. 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
<pre> <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> (...) </pre>	<p>Compute the new value of the fact "Chocolate_Box_Weight" using a formula.</p> <p>Here, a binder is used to define and evaluate the formula referred as "CalculateTotalWeight".</p>
<pre> <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> (...) </pre>	<p>Same, but in this case no binder is used and the C# expression is directly stored in the rule.</p> <p>See chapter 3.4 for more information on the syntax.</p>

3.2.7. Saliency and Weight

The engine estimates the saliency 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 saliency in order to calculate the overall implication weight:

$$\text{Weight} = (1 + \text{Priority}) * 100 + \text{Saliency}$$

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



3.3. 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.

3.3.1. 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.

3.3.2. Rulebase Adapters

The Inference Engine uses Rulebase adapters (i.e. classes implementing `NxBRE.InferenceEngine.IO.IRuleBaseAdapter` or `IExtendedRuleBaseAdapter`) 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.

3.3.3. Business Object Binders

The interface `NxBRE.InferenceEngine.IO.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 3.6 for a detailed view of execution time and to the API Documentation for the detail of the `IBinder` contract (see page 53).

Binders not only perform assertion of facts based on business objects but also evaluate function predicates. For example in the implication shown page 23, 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 too 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 an 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**.



3.4. 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 3.2.5),
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 3.2.6),

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.

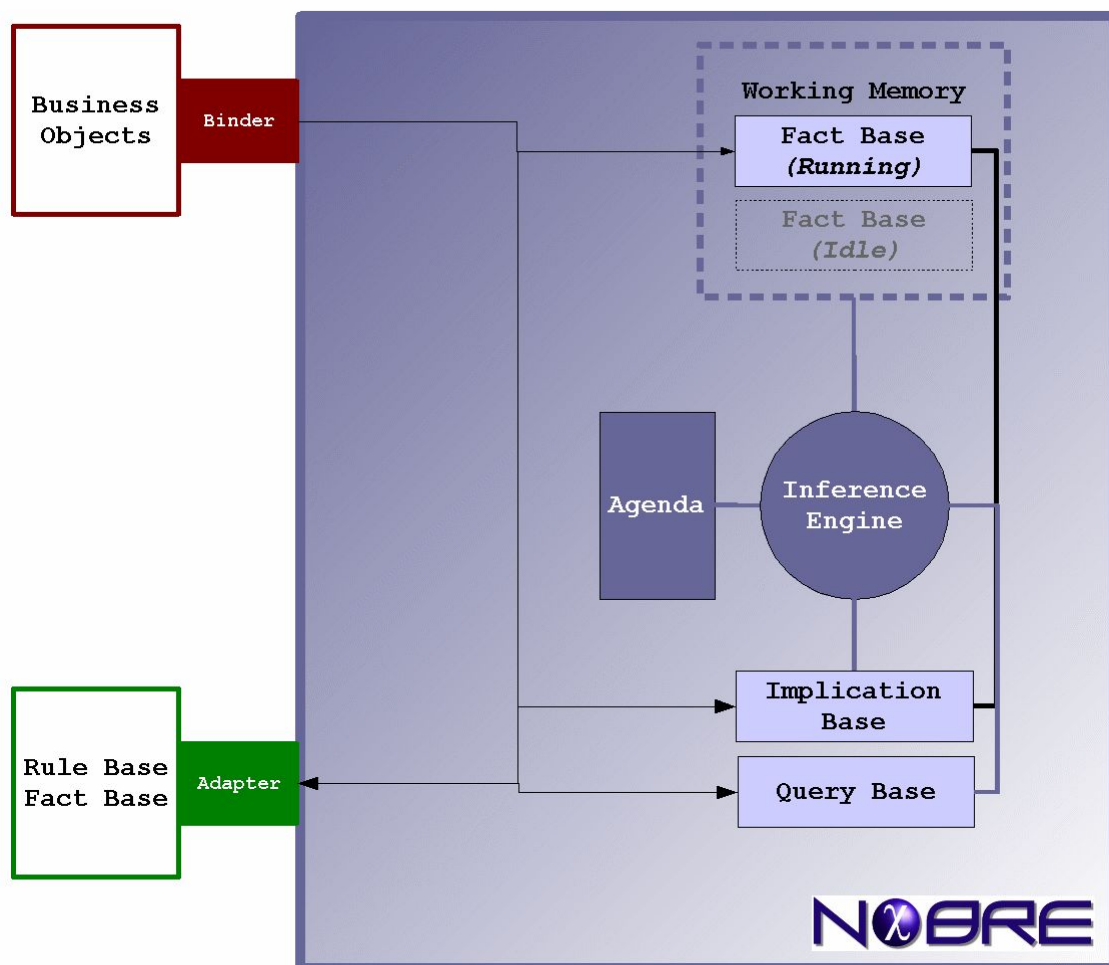
i C# expressions are compiled at first use: the predicates types are then frozen. This means that in a certain predicate, you must use the same type (or at least castable types) throughout your application. Failing to do so will result in casting errors at run-time.

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 3.2.6).



3.5. Engine



3.5.1. 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.

3.5.1.a. Fact Base

The Fact Base main index is a hierarchical storage of facts using their signature (fact relation type and number of predicates), predicates type, value and position as a composed key. Technically it is a:

```
IDictionary<string, IDictionary<Type, IDictionary<object, IDictionary<int,
ICollection<Fact>>>>>>
```

which gives a $O(5)$ cost for accessing any fact from one particular predicate.



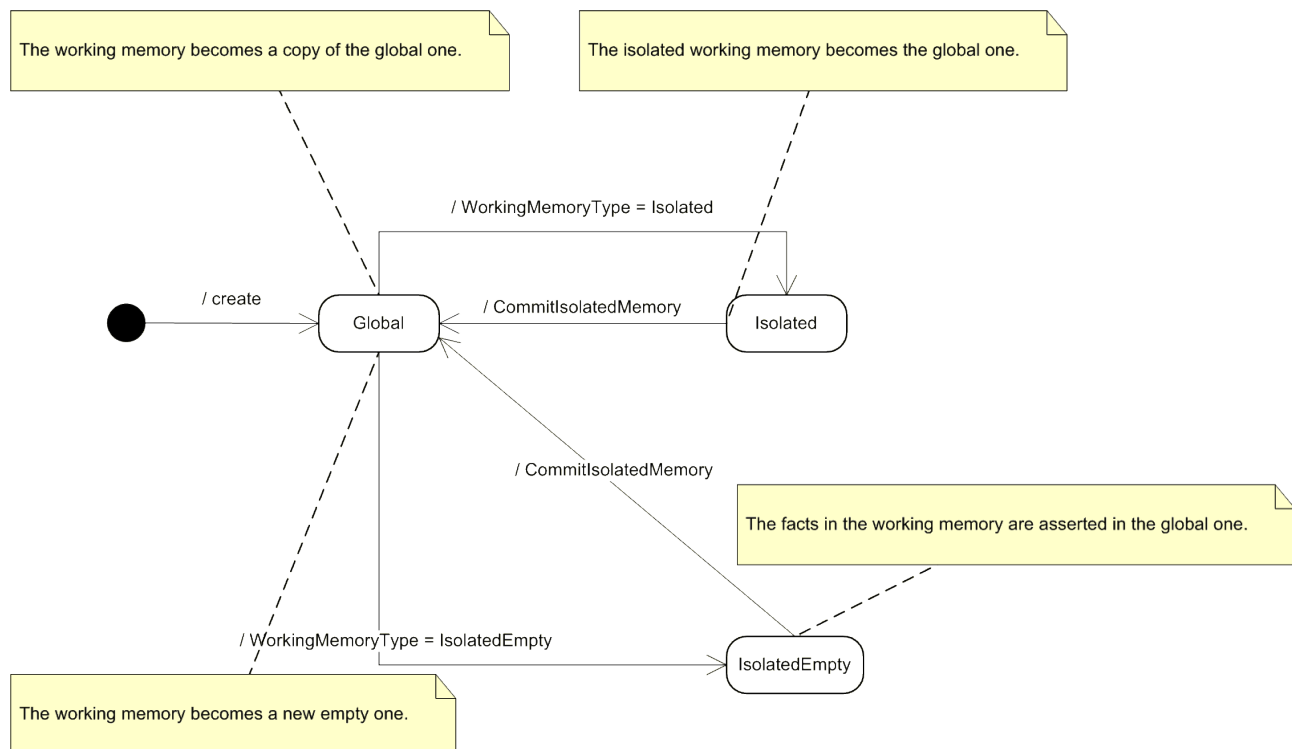
When trying to match a particular atom with facts stored in the fact base, the engine will first look for the smallest collection of fact matching a particular fixed predicate (like <Ind> or <Data>). On this smallest possible collection, the engine will then apply the remaining matching conditions, like other fixed predicate comparison, executing operator comparisons or function resolutions in order to produce the list of matching facts. With well discriminated facts (like facts having all a different ID), the cost of selecting one fact can be very small: $O(5)+n-1$, where n is the number of predicates (see chapter 3.11 for more tips on performance).

For critical systems, when loading facts in the engine, it is recommended to check that the asserted facts were all accepted by the Fact Base to ensure that no collision on a similar pre-existing fact occurred.

3.5.1.b. *Global, Isolated and IsolatedEmpty Modes*

When the Working Memory is in Global mode, all facts exist 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 deduced facts concern business objects undergoing an evaluation process.





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.

3.5.2. 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 3.2.7).

The Agenda decides what implications must be scheduled by analyzing the results of the previous iteration in the process cycle (see chapter 3.6). 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.

3.5.3. 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.

3.5.4. 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.



3.6. Execution

The process cycle of the Inference Engine (`NxBRE.InferenceEngine.IEImpl`) is very simple: basically it infers as long as new facts are deducted. If during an iteration all the deducted facts were already present in the Fact Base, then the process stops. There is also a limit 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 were 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 were asserted or retracted during point 7, perform a new iteration → 3.



3.7. 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.

3.7.1. 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  IIInferenceEngine 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).



3.7.2. 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  IIInferenceEngine 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 configurable time-out (see chapter 4). 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.

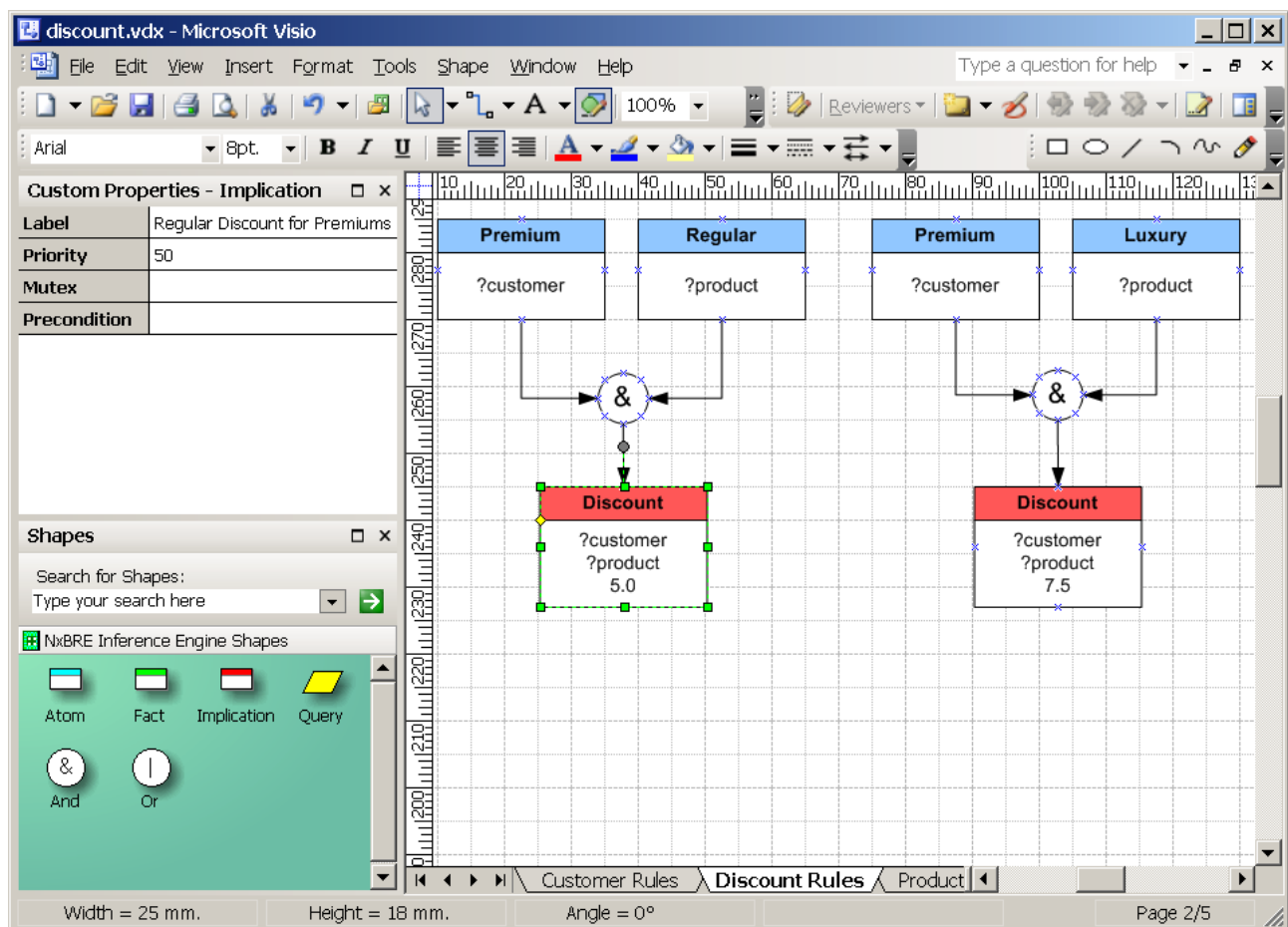


3.8. Microsoft Visio 2003 Adapter

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.

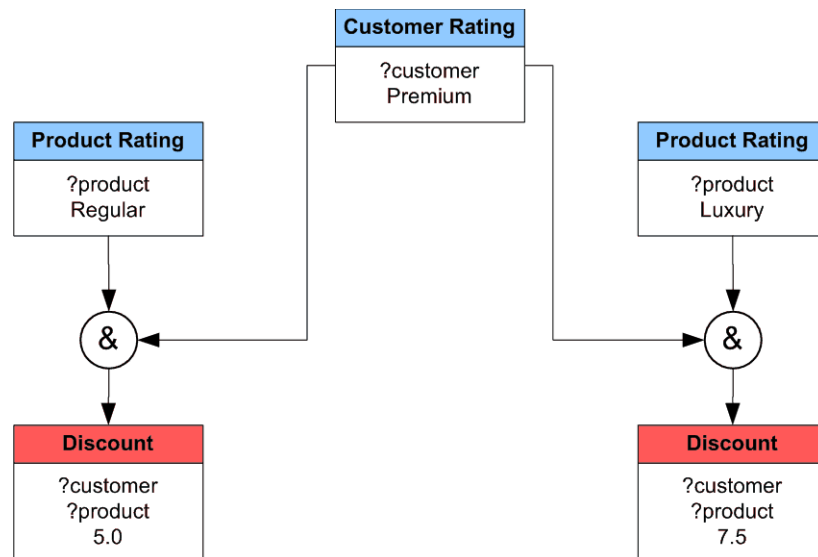
The immediate advantage is to allow developers to model in a unique environment (Visio) all their systems in UML, connecting use cases and other UML models to business rules expressed with **NxBRE's** stencil.



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.

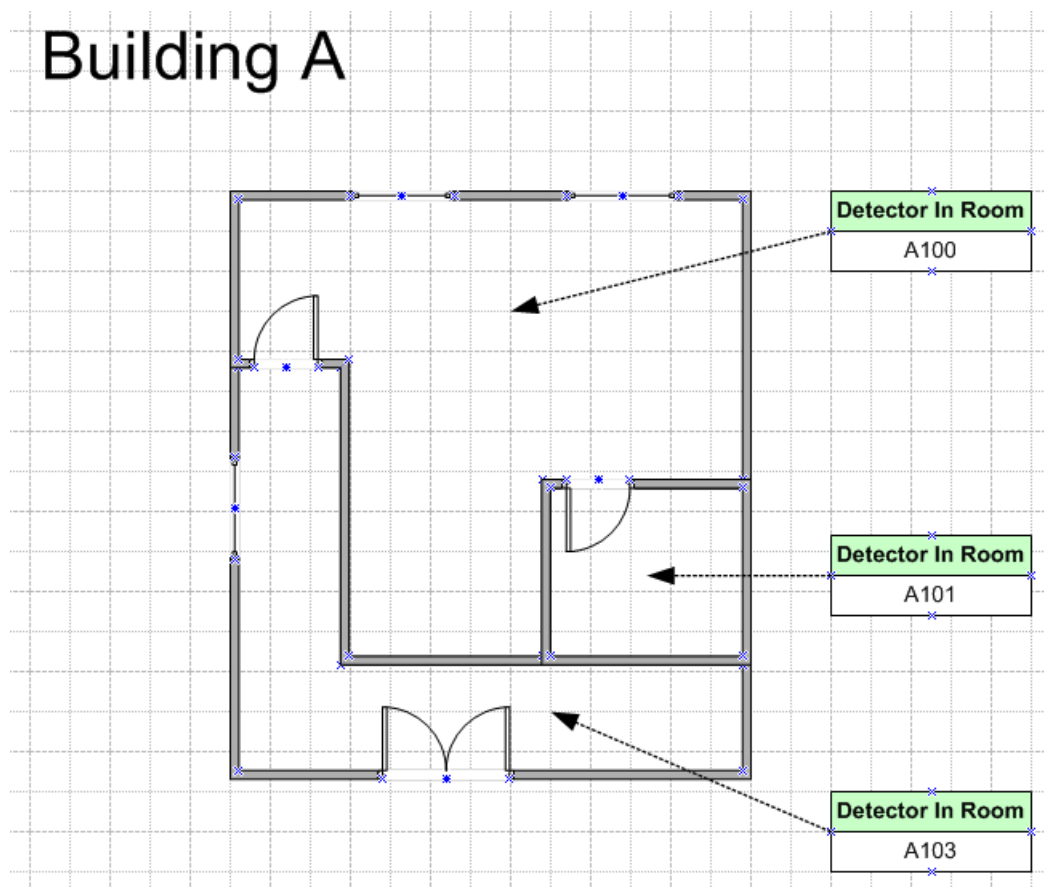



- **Atom syndication:** as shown above, it is possible to syndicate atoms that are used by different implications (or queries), limiting potential mistakes and increasing readability by reducing duplicated information.
- **Easy operators:** when transforming DataDiagramML to RuleML, the XSL-T takes care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

Operator in Visio	NxBRE Operator
>=	GreaterThanOrEqualTo
<=	LessThanOrEqualTo
<>	NotEquals
!=	
==	Equals
=	
<	LessThan
>	GreaterThan



- **Easy typed predicates:** by using a prefix like (**xs:int**) it is possible to type a predicate directly from the Visio rule base.
- **Easy named predicates:** using simple prefixes like (**?Size**) it is possible to name a predicate directly from the Visio rule base. Note that this prefix must always be first in the predicate description.
- **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.



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



3.9. 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
<pre> 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:GreaterThanOrEqualTo(100)} balance{?account, NxBRE:LessThanOrEqualTo(50)} -> testAandBorC{?account}; </pre>

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).



3.10. Registry

3.10.1. Concepts

The main goals of the *NxBRE.InferenceEngine.Registry.IRegistry* is to facilitate the loading of multiple rule bases by storing them in shared registry and to encourage a correct usage in a multi-threaded environment (as advocated in chapter 3.7).

Whatever the implementation is, the registry must exist as a single instance in the target application. If a static instance is not an option for your project, an elegant solution consists in using [Spring](http://springframework.net)⁴ to manage your objects graph and have the registry singleton instantiated and injected in your worker objects by this framework.

3.10.2. File Registry

NxBRE comes with one implementation of *IRegistry*, the *FileRegistry*. This implementation works with all rule and binder files stored in one particular folder: it monitors any change to these files and automatically reloads them without disturbing the application that uses the engine managed by the registry (provided the recommendations in chapter 3.7 have been followed!).

This registry is configured by an XML file constrained by the *nxbre-file-registry.xsd* schema file. By default, it looks for binder and rule files in the same folder where the configuration file is located, but another folder can be defined in the configuration file itself.

The following demonstrate a configuration file defining 5 engines and different rule files formats and binder types:

```
<?xml version="1.0" encoding="utf-8"?>
<FileRegistryConfiguration xmlns="http://nxbre.org/registry/file"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://nxbre.org/registry/file
        nxbre-file-registry.xsd">

    <Engine id="chocolatebox-binderless">
        <Rules file="chocolatebox-binderless.ruleml" format="RuleML09NafDatalog"/>
    </Engine>
    <Engine id="chocolatebox-ccb">
        <Rules file="chocolatebox.ruleml" format="RuleML09NafDatalog"/>
        <CSharpBinder file="chocolatebox.ruleml.ccb"
            class="NxBRE.Test.InferenceEngine.ChocolateBoxBinder"/>
    </Engine>
    <Engine id="events-test-vcb">
        <Rules file="events-test.ruleml" format="RuleML09NafDatalog"/>
        <VisualBasicBinder file="events-test.ruleml.vcb"
            class="NxBRE.Test.InferenceEngine.EventTestBinder"/>
    </Engine>
    <Engine id="chocolatebox-feb">
        <Rules file="chocolatebox.ruleml" format="RuleML09NafDatalog"/>
        <FlowEngineBinder file="chocolatebox.ruleml.xbre" type="BeforeAfter"/>
    </Engine>
    <Engine id="chocolatebox-binderless-visio">
        <Rules file="chocolatebox-binderless.vdx" format="Visio2003"/>
    </Engine>
</FileRegistryConfiguration>
```

4 see: <http://springframework.net> – using Spring in your applications is a good idea anyway!



3.11. Performance tuning

Here are several tricks that can help you making the most of **NxBRE** in term of performances.

3.11.1. Have strongly identified facts

If possible, each fact should have a unique identifier. Placing this identifier in the first position of the predicate list helps accessing this fact faster:

Definition: `DoB{CustomerId, CustomerDoB}`

Sample: `DoB{12345, "1971-01-02"}`

3.11.2. Use small facts

NxBRE has an access time to any single fact that depends on the number of predicates in this fact and not the amount of facts. Therefore it is better to break down a fact containing several predicates into several smaller facts.

Moreover, it is good practices to have facts always carry sense in a self explicit way: this is almost impossible with a big fact.

Consider the following comparison:

Definition: `Details{CustomerId, CustomerDoB, CostumerCountry, CustomerRating}`

Sample: `Details{12345, "1971-01-02", "CA", "Premium"}`

versus:

Definitions: `DoB{CustomerId, CustomerDoB }`
`Country{CustomerId, CostumerCountry }`
`Rating{CustomerId, CustomerRating}`

Samples: `DoB{12345, "1971-01-02"}`
`Country{12345, "CA"}`
`Rating{12345, "Premium"}`

Which approach is clearer? Remember that you will have to look at actual data (as shown in the sample)...

3.11.3. Use typed data and storage

For this, first leverage RuleML 0.9 Data elements that allow creating typed predicate both for facts defined in the rule file and for selection atoms in implications and queries.

Second, set the fact base to use strict typing only (see the configuration overview chapter 4): this reduces the amount of stored data and casting operations.



3.11.4. Order atoms in And blocks

NxBRE optimize the members of And and Or blocks by giving different priorities to atoms, negative atoms and nested logical blocks. But it does not optimize the order of matching the atoms patterns between themselves: thus, it is worth using your knowledge of the facts that will be stored in the working memory to sort atoms (standard ones, not negative ones) in a way that the ones producing the least results will be processed first.

Consider the following example:

```
BelongsTo{?Account, ?Transaction} AND Holds{?Customer, ?Account}
```

versus:

```
Holds{?Customer, ?Account} AND BelongsTo{?Account, ?Transaction}
```

The first condition will select all the transactions then filter out only the ones belonging to a particular account while the second condition will first select a particular account then select its transactions. Therefore, the latter is much better than the former.



4. Configuration

NxBRE is fully configurable via the standard .NET application configuration mechanism. The following table details these parameters.


AppSetting Key Name	Default	Description
<code>nxbre.iterationLimit</code>	1000	The maximum number of iteration to perform in one process cycle. If this limit is reached, the engine will throw an exception.
<code>nxbre.strictImplication</code>	False	Defines whether the engine should throw an exception in case an implication tries to assert a fact whose variable predicates have not all be resolved by the data returned by the atoms of the body.
<code>nxbre.lockTimeOut</code>	15000	The time-out in millisecond for acquiring a lock when hot swapping a rule base in multi-threaded environments.
<code>nxbre.abstractbinder.function.regex</code>	<code>^(?<1>\w+)\x28((?<2>[^\x28\x29]+),?)*\x29\$</code>	The regular expression used by default binders to estimate if the String content of an Individual represents a Function.
<code>nxbre.factBase.strictTyping</code>	False	Defines whether the fact storage should consider typed objects as equivalent to their String representation. If <code>StrictTyping</code> is set to true, they are will not be considered equivalent. This implies that to match typed values, <code><Data></code> elements instead of <code><Ind></code> must be used in the rulebase.
<code>nxbre.referenceLinkMode</code>	CurrentDomain	Defines the different strategies for adding references when on-the-fly compiling classes. Can be set programmatically via this static property: <code>NxBRE.Util.Compilation.ReferenceLinkMode</code>
<code>nxbre.cultureInfo</code>	en-US	Defines the format used by NxBRE when casting values. Can be set programmatically via this static property: <code>NxBRE.Util.Reflection.CULTURE_INFO</code>
<code>nxbre.embeddedResourcePrefix</code>	<code>NxBRE.Resources</code>	Defines the string that prefixes the resource names in the manifest. Should not be changed if building with VS.NET 2005/SharpDevelop2
<code>nxbre.unittest.inputfile</code>	<code>Q:/test.xbre</code>	Files used during unit testing. This configuration section is not necessary in production.
<code>nxbre.unittest.inputnative</code>	<code>Q:/discount.bre</code>	
<code>nxbre.unittest.identityxsl</code>	<code>Q:/identity.xsl</code>	
<code>nxbre.unittest.ruleml.inputfolder</code>	<code>Q:/</code>	Path to the Rulefiles folder of the distribution.
<code>nxbre.unittest.outputfolder</code>	<code>C:/Temp</code>	Ensure that you have write privilege on this folder.



5. Logging

NxBRE provides different trace sources the implementer can leverage for debugging or for monitoring particular behaviors. The tracing threshold of these sources are set by trace switches of the same name.

NB. These trace sources replace the legacy event based logging mechanisms of NxBRE 2.

 Please refer to the .NET framework SDK documentation for more information on using trace sources (`System.Diagnostics.TraceSource`) and switches (`System.Diagnostics.TraceSwitch`).

Here is the list of available sources and switches:

Source / Switch Name	Purpose
NxBRE.FlowEngine	Trace events emitted by the Flow Engine.
NxBRE.FlowEngine.RuleBase	Trace events emitted by rule base level operations (log, exceptions).
NxBRE.InferenceEngine	Trace events emitted by the Inference Engine.
NxBRE.Util	Trace events emitted by the utility classes.

Tracing is controlled via the standard configuration mechanisms and also by code via the utility class `NxBRE.Util.Logger` (turn to the API Documentation of NxBRE for more information).



6. API Documentation

The API Documentation should be bundled with the current document.

NB. The current version of NxBRE is stored in this static:

```
NxBRE.Util.Reflection.NXBRE_VERSION
```



7. 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](#).



8. Other engines

The following is not an exhaustive list of alternative engines for the .NET platform but should give you some hints on what is available out there.

8.1. Open source engines

There are much less open source engines in the .NET world than in the Java world. The good news is that mainstream Java engines are now being ported to .NET!

8.1.1. Drools DotNet

Though in beta when this guide is being written, Drools will deliver a full fledged RETE based rules engine to the .NET platform. The key feature differences with **NxBRE** are the possibility to define domain specific languages (DSL) to express the rules and the opt for a professional paid-for support.

Note that an Eclipse based editor that allows to edit the DSL-based rules is available.

8.1.2. Simple Rule Engine (SDSRE)

This is a lightweight forward chaining inference rule engine for .NET. It is considered *simple* because of the simplicity in writing and understanding the rules written in XML, but this simple engine can solve complex problems. It can be an alternative to **NxBRE** if none of the available rule formats are satisfying the user's needs.

8.2. Commercial engines

There are plenty of available business rules engines that you can purchase. On top of support and liability, your money will buy extra professional features that are not commonly found in the open source world like advanced rules editors or rules testing and staging features.

To list only a few, in no particular order:

- ILOG Rules
- Yasutech Quickrules
- Fair Isaac Blaze Advisor
- InRules