XSLT Code Generation and Java XPath Evaluation

T

|  |  |
| --- | --- |
| **Product Suite** | **TIBCO BusinessEvents** |
| **Product** | **TIBCO BusinessEvents** |
| **Product Version** | **5.1** |
| **Initial Author** | **Ryan Hollom** |

Copyright Notice

COPYRIGHT© 2008-2009 TIBCO Software Inc. This document is unpublished and the foregoing notice is affixed to protect TIBCO Software Inc. in the event of inadvertent publication. All rights reserved. No part of this document may be reproduced in any form, including photocopying or transmission electronically to any computer, without prior written consent of TIBCO Software Inc. The information contained in this document is confidential and proprietary to TIBCO Software Inc. and may not be used or disclosed except as expressly authorized in writing by TIBCO Software Inc. Copyright protection includes material generated from our software programs displayed on the screen, such as icons, screen displays, and the like.

Trademarks

Technologies described herein are either covered by existing patents or patent applications are in progress. All brand and product names are trademarks or registered trademarks of their respective holders and are hereby acknowledged.

Confidentiality

The information in this document is subject to change without notice. This document contains information that is confidential and proprietary to TIBCO Software Inc. and may not be copied, published, or disclosed to others, or used for any purposes other than review, without written authorization of an officer of TIBCO Software Inc. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

Content Warranty

The information in this document is subject to change without notice. THIS DOCUMENT IS PROVIDED "AS IS" AND TIBCO MAKES NO WARRANTY, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO ALL WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. TIBCO Software Inc. shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

For more information, please contact:

TIBCO Software Inc.  
3303 Hillview Avenue  
Palo Alto, CA 94304  
USA

Table of Contents

[XSLT Code Generation and Java XPath Evaluation 1](file:///C:\Users\suresh\Desktop\BE-5%201%20XSLT%20Code%20Gen%20and%20Java%20XPath%20evaluation.docx#_Toc315868260)

[Change Log 4](#_Toc315868261)

[1 Introduction 5](#_Toc315868262)

[1.1 Purpose 5](#_Toc315868263)

[1.2 Scope 5](#_Toc315868264)

[1.3 Audience 5](#_Toc315868265)

[2 Acronyms and Vocabulary 5](#_Toc315868266)

[3 XPath Evaluation with Java Obects 6](#_Toc315868267)

[3.1 Background 6](#_Toc315868268)

[3.2 Problem 7](#_Toc315868269)

[3.3 Solution 7](#_Toc315868270)

[3.4 XPath Evaluation Process (detailed) 8](#_Toc315868271)

[3.4.1 Expressions 8](#_Toc315868272)

[3.4.2 Evaluation 9](#_Toc315868273)

[3.4.3 Node Resolver 9](#_Toc315868274)

[3.5 Constant XPath Expressions 10](#_Toc315868275)

[3.6 Performance Results 11](#_Toc315868276)

[4 Java Code Generation for XSLT Transformations 12](#_Toc315868277)

[4.1 Background 12](#_Toc315868278)

[4.2 Solution 13](#_Toc315868279)

[4.3 Example 14](#_Toc315868280)

[5 Results 15](#_Toc315868281)

[5.1 BE Engine Performance 15](#_Toc315868282)

[5.2 Apache JXPath 19](#_Toc315868283)

[6 Summary 19](#_Toc315868284)

# Change Log

.

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Name | Description |
| 01/27/2012 | 1.0 | Ryan Hollom | Initial Document Creation |
| 01/29/2012 | 1.0.1 | Suresh | Update TOC. |
| 1/31/2012 | 1.0.2 | Ryan Hollom | Update order, expand based on Suresh’s comments |
| 2/1/2012 | 1.0.3 | Ryan Hollom | Added performance/execution results |
| 2/1/2012 | 1.0.4 | Suresh | Merged Ryan’s comments and performance. |
| 2/1/2012 | 1.0.5 | Ryan Hollom | Added XSLT Transformation performance info |

# Introduction

## Purpose

The purpose of this document is to discuss an alternative approach to performing XSLT Transformations and XPath evaluation inside of BE Projects.

## Scope

This scope of this document is to provide detailed information regarding a java code generative approach to XSLT Transformations, as well as to provide information regarding XPath evaluation using domain specific Java object.

## Audience

Developers, Architects, QA, Support, and Product Managers

# Acronyms and Vocabulary

The table below describes some of the Acronyms used in this documentation

1. BE – TIBCO BusinessEvents.
2. XPath – XML Path Language (see <http://www.w3.org/TR/xpath/>)
3. XLST – Extensible Stylesheet Language (see [http://www.w3.org/TR/xslt](http://www.w3.org/TR/xslt/)/)
4. Apache jxpath : [commons.apache.org/jxpath](file:///C:\dev\be\5.1\docs\design%20spec\commons.apache.org\jxpath\)/
5. ANTLR – ANother Tool for Language Recognition

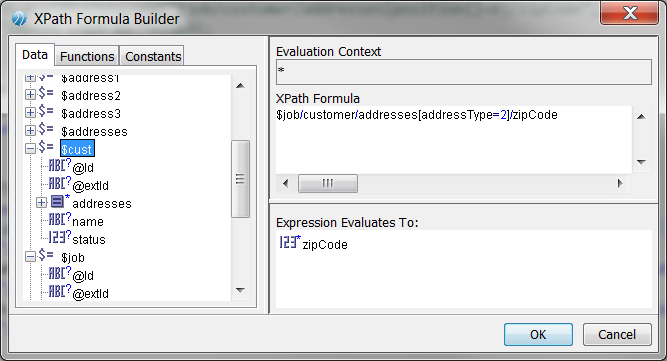
# XPath Evaluation with Java Obects

## Background

TIBCO BusinessEvents(BE) provides developer to access its internal structures(facts) as Concepts and Events. A instance of concept is represented as Java Object instance of type “Concept”. Similarly an Event is point in time data and is represented as java object instance of “SimpleEvent” type. But they are all java objects as far as the java heap is concerned.

BE provides a seamless access of these object as either java object or XML DOM objects. Furthermore it allows for the use of XML Path Language (XPath) expressions inside of business rules. These expressions can be used to calculate primitive values (integers, Strings, Booleans, etc), as well as setting individual property values of output objects inside of the BE mapper (see Section 4).

XPath expressions are composed via an XPath Formula Builder inside of BE:



The XPath formula can use local variables in the scope of the BE Rule/Rule Function to perform the XPath evaluation. These local variables can be primitive types, or they can be complex types such as Event, Concept, etc. The current XPath evaluation process is seen in Figure 1:

Figure 1

## Problem

In the current process, the conversion of the XPath expression arguments from their Java object representation to DOM object (XiNode) **is an extremely expensive one**, particularly for complex data types. Inside of BE, a Concept element can contain several levels of nested data in the form of Contained Concepts/Concept References. The result is that every time the Concept is used in an XPath expression, even if it is for one single primitive property, the entire Concept data structure is converted into an XiNode to be processed by the XQueryExpr.

Caching the DOM graph for a java object would be one solution. BE does it for Event types, however Concepts are mutated on every execution cycle, there invalidating the cache. Cache actually introduces more overheads, and race-conditions on a multi-threade environment.

## Solution

The solution is to provide an XPath evaluator that works directly against the incoming Java object graph to avoid the expensive conversion of the data to dom node. This TIBCO Java XPath evaluator (JXPath) is to be compliant with the 1.0 specification of the XPath Language, outlined at <http://www.w3.org/TR/xpath/>. The JXPath evaluator supports all standard XPath functions (count, number, concat, etc).

For example, the XPath expression $job/customer/addresses[addressType=2]/zipCode works with a complex data object with several levels of nested data. Ultimately, this XPath expression is traversing the $job object’s structure to find a particular zipCode from a specific address. Converting the $job object into an XiNode is extremely expensive, especially if there are a large number of nested address elements.

Figure 2

In the proposed solution, the conversion of the Java objects to XiNode is avoided completely. Instead, the NodeResolver is responsible for resolving the variables used in the XPath expression. The NodeResolver is also responsible for traversing the structure of the domain object data model. In the case of BE, the NodeResolver works with Concept and Event objects to access property definitions, contained concept information, etc. This can be expanded to fit any Bean model or any interfaces that the Java Object knows for accessing its properties.

## XPath Evaluation Process (detailed)

The first step in the evaluation is to compile the XPath expression into a javax.xml.xpath.XPathExpression (JXPathExpression). This is done by first parsing the XPath expression with an ANTLR generated parser written for the XPath language. The output of the parsing step is an Abstract Syntax Tree (AST), which is then used to create a com.tibco.jxpath.Expression object, which is wrapped by JXPathExpression.

### Expressions

There are several Expression types, as described below:

|  |  |
| --- | --- |
| BinaryOperation | Operation which takes a left and right side, including And, Div, Equals, Dt, Dte, Lt, Lte, Minus, Mod, Mult, NotEquals, Or, Plus |
| NaryOperation | Operation that takes operates on a List of expression elements, which includes FunctionCall |
| PathExpression | Contains a list of Expression elements, or steps, each of which gets evaluated sequentially. The context node of the current step is determined by the previous step in the PathExpression |
| PredicateExpression | Expression which filters a set of nodes, and includes FilterExpression and NamedAxisStep |
| StepAxisExpression | Represents a separator between the steps of a PathExpression, and can operate by either navigating down the structure of the context node (child nodes) or up the structure of the context node (parent node) |
| UnaryOperation | Operation with a single argument, including Neg |
| Variable | Refers to a variable to be resolved by the NodeResolver. The variable resolves to an XObject |
| XObject | This object can either be a list of objects (XObjectList), a primitive type (XBoolean, XNumber, XString), or it can wrap domain specific objects (XObjectWrapper). |

### Evaluation

The output of the compilation process is a JXPathExpression, which can then be evaluated. The evaluation process requires a javax.xml.xpath.XPathVariableResolver, as well as the expected output/return type. In the BE case, there is a JXPathBEVariableResolver responsible for navigating the structure of the context object.

At the start of the JXPathExpression evaluation, an XPathContext is created, which holds information regarding the state of the current evaluation. This interface contains the following APIs:

**public** **interface** XPathContext {

NodeResolver getNodeResolver();

XPathFunctionResolver getFunctionResolver();

XObject getCurrentContextNode();

**boolean** isAbbreviatedStep();

**int** getCurrentContextPosition();

**int** getCurrentContextCount();

}

This interface is implemented by JXPathContext, and holds references to a NodeResolver (JXPathBEVariableResolver), the XPathFunctionResolver, and the current context node, which is of type com.tibco.jxpath.XObject. This object can either be a list of objects (XObjectList), a primitive type (XBoolean, XNumber, XString), or it can wrap domain specific objects (XObjectWrapper).

### Node Resolver

It is the responsibility of the NodeResolver to know how to traverse the structure of the domain objects. The NodeResolver API is as follows:

**public** **interface** NodeResolver **extends** XPathVariableResolver {

/\*\*

\* Get the Xobject for Named axis w.r.t the parentContext object.

\* **@param** context - This can be variable value returned from the call to resolveVariable or from the call to resolveNamedAxis.

\* **@param** qName - The qName from the xpath.

\* **@param** abbr - This tells if we have to look recursively in the context

\* **@param** axisName - the name of the axis

\* **@return**

\*/

Object getChild(XObject context, QName qName, **boolean** abbr, AxisName axisName);

/\*\*

\* Does this context object have children.

\* **@param** context

\* **@return**

\*/

**boolean** hasChildren(XObject context);

/\*\*

\*

\* **@param** context

\* **@return** the count of children for the context looked At.

\*/

**int** count(XObject context);

/\*\*

\* Get the Child at position specified for context object.

\* **@param** context

\* **@param** pos

\* **@return**

\*/

XObject getChild(XObject context, **int** pos);

/\*\*

\* Return the QName for this Context Node.

\* **@param** ctxNode

\* **@return**

\*/

QName name(XObject ctxNode);

/\*\*

\*

\* **@param** contextNode

\* **@param** id

\* **@return**

\*/

XObject getChildById(XObject contextNode, String id);

}

This approach decouples the JXPath library from any specific implementation. In the case of BE, the JXPathBEVariableResolver (which implements NodeResolver) has knowledge about BE entities (Concepts, Events, Properties, etc), and can thus navigate the structure of these elements in a generic fashion such that it fits into the JXPath paradigm. There is no reference to anything BE specific in the core JXPath APIs, and so this library can be reused across any number of domains. This is an important note to stress, as there are a great deal of other products that can benefit from this library simply by implementing their own domain specific NodeResolver. Inside of the JXPath library, the NodeResolver is used by PredicateExpression (and its subclasses) to delegate the traversal of the XPath expression to the domain specific implementation.

By working directly with the domain model Java object, inefficient conversions are avoided, memory is saved

## Constant XPath Expressions

In some cases, it is possible to directly generate the Java code for these individual XPath expressions. This is true for constant expressions, such as direct property access of a Concept property. For example, $cust/status simply accesses the ‘status’ property of the $cust Concept object. In these cases, the resulting Java code can be generated directly rather than using JXPath. This avoids the extra steps of compiling the XPath expression into a JXPathExpression, initializing the evaluator, etc.

For example, if the $cust/status field is mapped to the ‘outputStatus’ property, the generated Java would be:

$output.setPropertyValue(

"outputStatus",

((com.tibco.cep.runtime.model.element.PropertyAtomInt) $3zcust

.getProperty("status")).getInt());

## Performance Results

The following 3 expressions were evaluated 300000 times sequentially

expr[0] = xpath.compile("/job/customer/addresses[addressType=1]");

expr[1] = xpath.compile("/job/customer/addresses[2]");

expr[2] = xpath.compile("/job/customer/addresses[position()=2]");

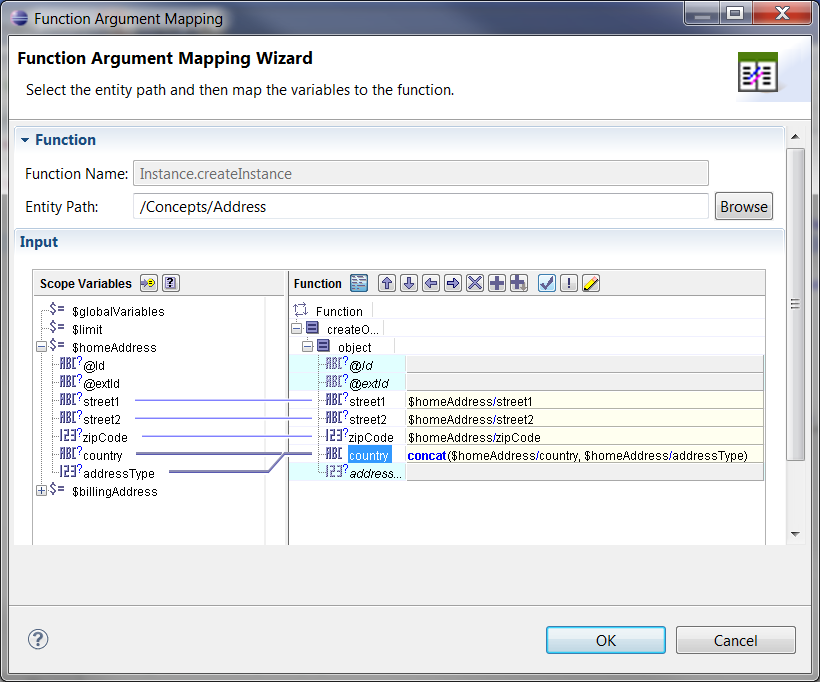
A total of 1M evaluations

|  |  |  |
| --- | --- | --- |
| **TIBCO JXPATH (JAVA OBJECT)** | **STANDARD XPATH (JVM provided or Apache)** | **TIBCO JXPATH (DOM NODE)** |
| **522 ms** | 129832 ms | 1696 ms |

# Java Code Generation for XSLT Transformations

## Background

There exists a feature in TIBCO BusinessEvents (hereafter BE) in which users can create BE elements, called Concepts and Events, by using XSLT transformations. This is done by mapping incoming data to the properties of the created elements by using a visual mapper tool inside of TIBCO BusinessEvents Studio:



Individual property assignments are in the form of XPath expressions. The current XSLT transformation flow is shown in Figure 3:

Figure 3

During runtime, the incoming domain model data objects are converted into an XML object node (XiNode) and are passed to a transformation engine that uses the mapping information provided to create the output object. All attributes of the incoming domain model data objects are captured in the XiNode to provide the transformation engine the information needed to create the output object. For the BE use case, the output object is a Java object. As such, it is often the case where the incoming domain model data object is a plain Java object, whose properties are used to set the properties of the output Java object.

Conversion from the incoming data to an XiNode is expensive, particularly for complex data. BE allows for highly complex nested data in the form of Contained Concepts or Concept References. If the incoming data is in this form, the conversion to XiNode will convert all levels of the data, potentially several levels deep, even if only a small subset of this information is needed to perform the transformation. This conversion is done each time a mapping is done inside of a BE project. For high volume projects, this can translate to thousands of mappings per second. As such, avoiding any unnecessary or inefficient conversion would result in large performance gains for BE applications.

## Solution

The current implementation that converts the incoming Java object into an XiNode, passes that node to a transformation engine, and outputs a Java object is highly inefficient. This is particularly true for complex data. Since the BE project is generated into Java code, it is far more efficient to directly generate the Java to create the output Java object. In the case of constant XPath expressions (sec. 3.4), the Java code can be generated directly to set the output properties by using standard get APIs from the incoming Java data object and set APIs on the output Java object. For other XPath expressions, the JXPath evaluator can be used to calculate the value of the XPath expression and set the output property value appropriately. By doing this, the conversion to XiNode is avoided, and only the necessary information from the incoming data is accessed.

Figure 4

With the proposed solution, there is no need to convert the arguments used to XiNode, as the JXPath evaluator works directly with the domain (Java) objects, as described in section 3.

## Example

For the mapping shown in section 4.1, a BE Concept of type Concepts.Address is being created. The properties of that Concept are set from an incoming ‘homeAddress’ object, which is also of type Concepts.Address. In this simple example, the mapping information can be used to directly generate the java to set the output properties based on the incoming data:

com.tibco.cep.runtime.session.RuleSession session = com.tibco.cep.runtime.session.RuleSessionManager

.getCurrentRuleSession();

com.tibco.cep.kernel.service.ObjectManager objectManager = session

.getObjectManager();

be.gen.Concepts.Address $temp = **null**;

java.lang.Class clz = be.gen.Concepts.Address.**class**;

**long** id = session.getRuleServiceProvider().getIdGenerator()

.nextEntityId(clz);

$temp = **new** be.gen.Concepts.Address(id);

$temp.setPropertyValue(

"street1",

((com.tibco.cep.runtime.model.element.PropertyAtomString) $3zhomeAddress

.getProperty("street1")).getString());

$temp.setPropertyValue(

"street2",

((com.tibco.cep.runtime.model.element.PropertyAtomString) $3zhomeAddress

.getProperty("street2")).getString());

$temp.setPropertyValue(

"zipCode",

((com.tibco.cep.runtime.model.element.PropertyAtomInt) $3zhomeAddress

.getProperty("zipCode")).getInt());

$temp.setPropertyValue(

"country",

com.tibco.be.functions.xpath.JXPathHelper

.evalXPathAsString(

"concat($homeAddress/country, $homeAddress/addressType)",

**new** String[] { "homeAddress", "homeAddress" },

**new** Object[] {

com.tibco.cep.util.CodegenFunctions

.box($3zhomeAddress),

com.tibco.cep.util.CodegenFunctions

.box($3zhomeAddress) }));

com.tibco.be.functions.object.ObjectHelper.createStateMachine(session,

(com.tibco.cep.runtime.model.element.impl.ConceptImpl) $temp,

**true**, **false**);

session.assertObject($temp, **false**);

Note that the mapping of the ‘country’ field is done by calling the JXPath library to evaluate the XPath expression. The other XPath expressions are treated as constant expressions, and the Java code is generated directly.

With this approach, no conversion to XiNode is needed, thereby saving processing time, as well as memory. For highly complex data, this has shown to reduce processing time by as much as 95%.

In addition, mappings are stored inside of BE Rules and Rule Functions (and therefore the generated Java code) as a Java String. For complex mappings, this String can be several thousand bytes or more. Inside of the JVM, this String is then stored inside of the Java heap, consuming large amounts of heap space. By directly generating the Java for the mapping, there is no need to insert the mapping String into the generated code, saving additional memory while running the BE project.

# Results

## BE Engine Performance

To test the performance gains, a sample BE project was created with a relatively complex hierarchy of BE Concepts, shown in Figure 5 here:

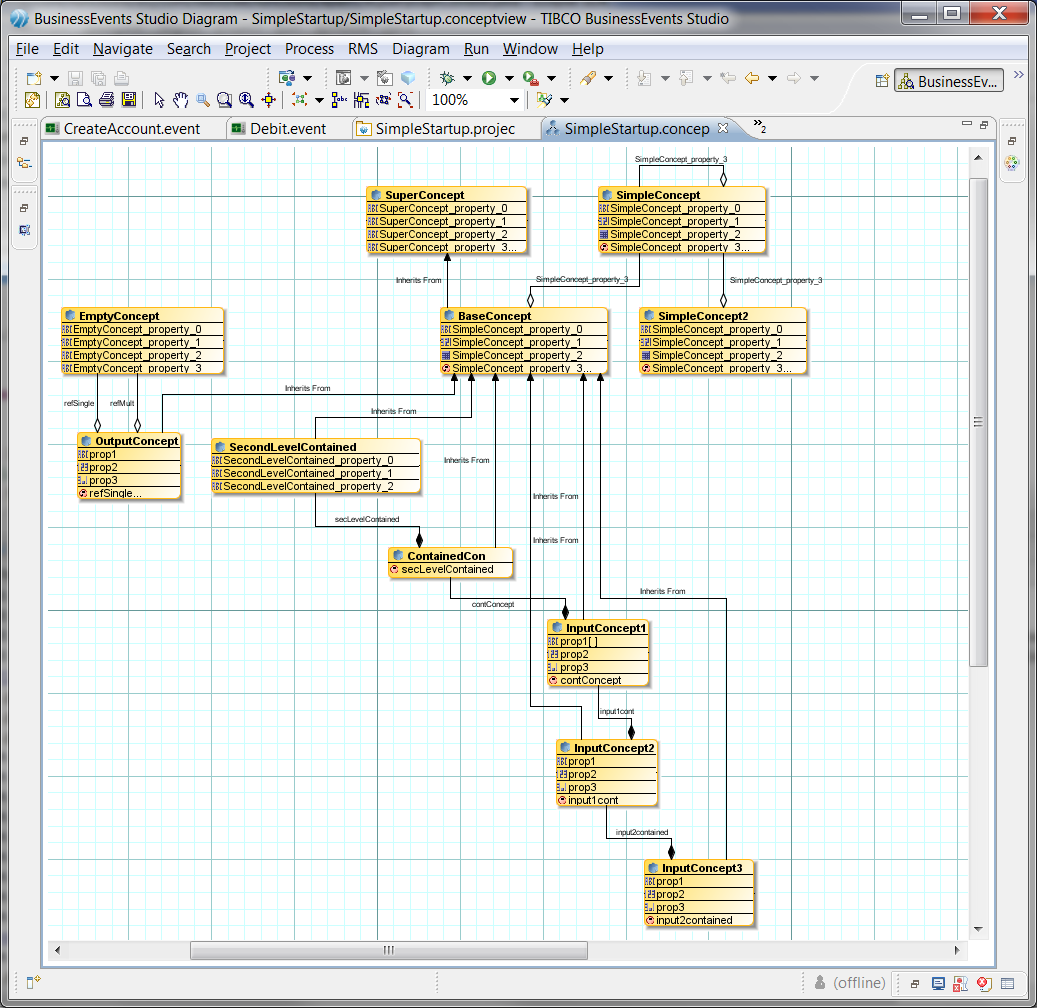


Figure 5

The base SuperConcept element contains 50 properties, and therefore all sub-Concepts also contain a minimum of 50 properties. Using this project, an instance of InputConcept3 was created in a BE rule (via its Ontology function). From that instance, the following XPath expressions were evaluated using the existing XPath evaluator, and then the same XPath expressions were evaluated using JXPath. Each group of XPath expressions were evaluated 5000 times (for a total of 50,000 evaluations). The results can be seen in Figures 6 and 7 below

**Concepts**.**InputConcept3** ic3= …

$ic3/input2contained/input1cont/contConcept/secLevelContained/SuperConcept\_property\_0

concat($ic3/input2contained/input1cont/contConcept/secLevelContained/SuperConcept\_property\_0, 1)

concat($ic3/input2contained/input1cont/contConcept/secLevelContained/SuperConcept\_property\_0, 2)

concat($ic3/input2contained/input1cont/contConcept/secLevelContained/SuperConcept\_property\_0, 3)

concat($ic3/input2contained/input1cont/contConcept/secLevelContained/SuperConcept\_property\_0, 4)

concat($ic3/input2contained/input1cont/contConcept/secLevelContained/SuperConcept\_property\_0, 5)

concat($ic3/input2contained/input1cont/contConcept/secLevelContained/SuperConcept\_property\_0, 6)

concat($ic3/input2contained/input1cont/contConcept/secLevelContained/SuperConcept\_property\_0, 7)

concat($ic3/input2contained/input1cont/contConcept/secLevelContained/SuperConcept\_property\_0, 8)

concat($ic3/input2contained/input1cont/contConcept/secLevelContained/SuperConcept\_property\_0, 9)

concat($ic3/input2contained/input1cont/contConcept/secLevelContained/SuperConcept\_property\_0, 10)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Std Deviation |
| XPath (in ms) | 20014 | 19881 | 20351 | 20188 | 19944 | 20075.6 | 192.00 |
| JXPath (in ms) | 250 | 259 | 249 | 276 | 259 | 258.6 | 10.83 |
| % Performance gain | 98.75 | 98.70 | 98.78 | 98.63 | 98.70 | 98.71 | 0.06 |

Figure 6

As can be seen in Figure 6, using JXPath for complex data traversal saves 98.71% of processing time. This is clearly a significant performance gain, though perhaps an uncommon use case. More often, the XPath expression is accessing a simple property of a complex data structure. Rather than traversing several levels of nested data, the above XPath expressions were modified to access a top level property, and the evaluation loop was run again. The results can be seen in Figure 7.

$ic3/SuperConcept\_property\_0

concat($ic3/SuperConcept\_property\_0, 1)

concat($ic3/SuperConcept\_property\_0, 2)

concat($ic3/SuperConcept\_property\_0, 3)

concat($ic3/SuperConcept\_property\_0, 4)

concat($ic3/SuperConcept\_property\_0, 5)

concat($ic3/SuperConcept\_property\_0, 6)

concat($ic3/SuperConcept\_property\_0, 7)

concat($ic3/SuperConcept\_property\_0, 8)

concat($ic3/SuperConcept\_property\_0, 9)

concat($ic3/SuperConcept\_property\_0, 10)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Std Deviation |
| XPath (in ms) | 19840 | 19835 | 20097 | 19992 | 19849 | 19922.6 | 117.42 |
| JXPath (in ms) | 174 | 181 | 169 | 207 | 180 | 182.2 | 14.69 |
| % Performance gain | 99.12 | 99.09 | 99.16 | 98.96 | 99.09 | 99.09 | 0.07 |

Figure 7

In the common use case of simple property access, the performance gain is even greater than it is for complex ones. Altering the above test so that it only ran once (rather than 5000 times) still shows significant savings, as shown in Figure 8

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Std Deviation |
| XPath (in ms) | 39 | 40 | 40 | 39 | 39 | 39.4 | 0.55 |
| JXPath (in ms) | 3 | 2 | 3 | 3 | 2 | 2.6 | 0.55 |
| % Performance gain | 92.31 | 95.00 | 92.50 | 92.31 | 94.87 | 93.40 | 1.41 |

Figure 8

For arithmetic expressions, there is also a significant performance improvement over the existing library. The expression

56 + (234 - 45) div 345 \* 235 \* 56 + (234 - 45) div 345 \* 235

was evaluated 50,000 times, and the results can be seen in Figure 9:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Math | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Std Deviation |
| XPath (in ms) | 442 | 444 | 457 | 466 | 440 | 449.8 | 11.23 |
| JXPath (in ms) | 129 | 127 | 131 | 129 | 129 | 129 | 1.41 |
| % Performance gain | 70.81 | 71.40 | 71.33 | 72.32 | 70.68 | 71.32 | 0.64 |

Figure 9

For simple models, there is still a non-trivial performance gain. For the model shown in Figure 10, the XPath expressions shown below were evaluated 50,000 times. The results can be seen in Figure 11.

XPath expressions:

$job/customer/name

$job/customer/status

concat($job/customer/name,$job/customer/status)

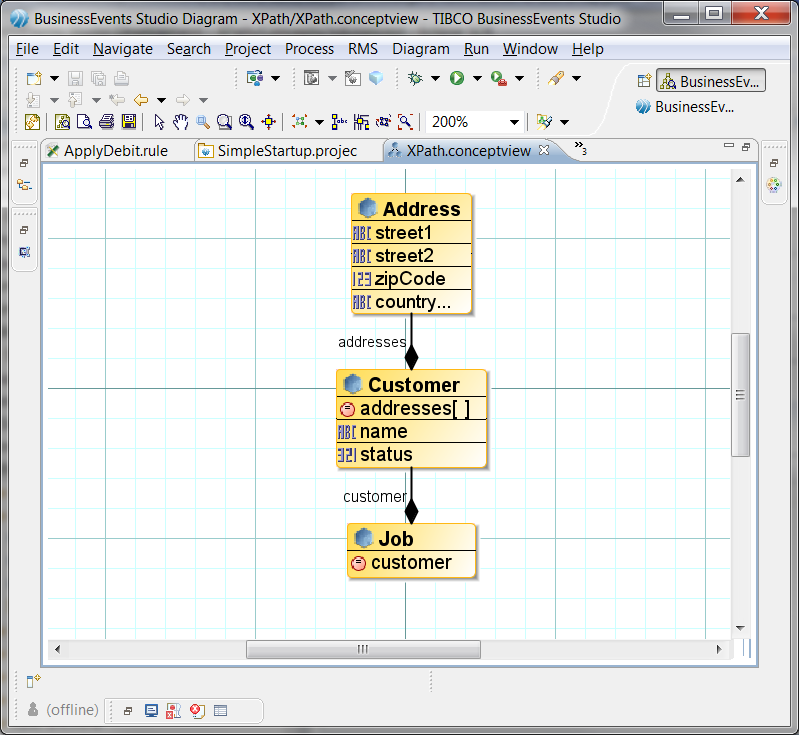


Figure 10

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Std Deviation |
| XPath (in ms) | 379 | 387 | 377 | 390 | 388 | 384.2 | 5.81 |
| JXPath (in ms) | 201 | 204 | 197 | 208 | 195 | 201 | 5.24 |
| % Performance gain | 46.97 | 47.29 | 47.75 | 46.67 | 49.74 | 47.68 | 1.22 |

Figure 11

## Apache JXPath

There is a library available from Apache which defines a simple interpreter of XPath, available in the org.apache.commons.jxpath package (see <http://commons.apache.org/jxpath/index.html>). This library applies XPath expressions to graphs of objects of all kinds, including JavaBeans. This library was compared to the BE JXPath evaluator, using simple POJOs (Plain Old Java Objects) to evaluate the following XPath expressions:

$job/customer/addresses[@addressType=1]

$job/customer/addresses[2]

$job/customer/addresses[position()=2]

The above XPath expressions were evaluated 1,000,000 times, and the results can be seen in Figure 12:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Std Deviation |
| Apache JXPath (in ms) | 7408 | 6746 | 6846 | 6854 | 6839 | 6938.6 | 266.03 |
| JXPath (in ms) | 1862 | 1492 | 1775 | 1768 | 1537 | 1686.8 | 162.37 |
| % Performance gain | 74.87 | 77.88 | 74.07 | 74.20 | 77.53 | 75.69 | 1.85 |

Figure 12

For the use case of simple POJOs (non-BE objects), the BE JXPath evaluator resulted in a 4x reduction in processing time. Therefore, BE JXPath is a valuable tool that can be leveraged across many different domains.

## XSLT Transformations

Using the complex model as shown in Figure 5, the existing XSLT transformation engine was compared to the direct Java generation approach as described in Section 4. For this test, 8 BE Concepts were created 1000 times, for a total of 8000 Concepts. This was done by using the Instance.createInstance method to map a small number of properties from incoming Concepts to the output Concepts. The results can be seen in Figure 13:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Create instance | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Std Deviation |
| Existing | 7070 | 7215 | 7091 | 7191 | 7018 | 7117 | 83.32 |
| Direct code gen | 269 | 277 | 273 | 280 | 275 | 274.8 | 4.15 |
| % Performance gain | 96.20 | 96.16 | 96.15 | 96.11 | 96.08 | 96.14 | 0.05 |

Figure 13

The table shows a significant reduction in object creation time when directly generating the Java code to create the object and set its properties. The same gain can be seen when running the loop a single time (~95% or a 20x reduction in processing time).

The same process was applied for the simple model shown in Figure 10. The results of which can be seen in Figure 14.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Create instance simple | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average | Std Deviation |
| Existing | 289 | 290 | 291 | 289 | 286 | 289 | 1.87 |
| Direct code gen | 152 | 154 | 154 | 154 | 153 | 153.4 | 0.89 |
| % Performance gain | 47.40 | 46.90 | 47.08 | 46.71 | 46.50 | 46.92 | 0.35 |

Figure 14

In a best case scenario, the direct java code generation approach results in a 20x reduction in processing time (~96%), and even in a simple case, there is still a 2x reduction in processing time.

# Summary

In summary, a combination of the above approaches (direct java code generation and Java based XPath evaluation) provides a valuable performance gain for the BE engine. Further, the Java based XPath evaluation has been implemented in an abstract manner so that it is reusable across other domains (see section 5.2), and can achieve valuable performance gains as BE has with its implementation.