

XPath API

XPath API

Contents

- [Introduction](#)
- [XPath basics](#)
- [XPath functions reference](#)
 - [Basic XPath functions:](#)
 - [Extended XPath functions](#)
 - [Language-specific XPath functions](#)
- [XPath API](#)
- [Extending XPathRule](#)

Introduction

[XPath](#) is a language for searching in trees (typically XML trees, but it could be applied to AST as well).

The simplest way to code a Kiuwan rule is with an XPath rule that expresses, using XPath notation, the condition that AST nodes must match to be considered violations of the rule. An XPath rule is declarative and does not need to be programmed, but requires a certain knowledge of the AST. You may use the **Rule Developer** to parse the AST for sample input code and execute XPath expressions against the parsed AST (XPath operates on both low-level and high-level ASTs).

For example, the following XPath expression detects loops without initialization nor update, that may be replaced by easier-to-understand while loops (first predicate admit for(;) loops, while last predicate excludes "for each" loops):

```
//ForStatement
[count(*)>1]
[not(ForInit)]
[not(ForUpdate)]
[not(LocalVariableDeclaration and Expression and Statement)]
```

To test such XPath expressions, you may use the Rule Developer tool:

[blocked URL](#)

XPath is adequate for simple rules. Typically, naming conventions could be easily verified using XPath expressions.

For example, if we are looking for the usages of `System.gc()` in Java code, the rule may use the XPath expression:

```
//Name[@Image='System.gc' or @Image='java.lang.System.gc']
```

NOTE: the XPath attribute axis (`@property`) fetches an AST node property (getter returning a primitive type). For example, `@Image` invokes the `getImage()` method on context AST nodes.

XPath-based rules could be implemented configuring `com.als.core.rule.XPathRule`, passing proper XPath expression in `xpath` rule property.

XPath basics

An XPath path expression uses steps and predicates (enclosed between square brackets `[` and `]`) to select nodes in a tree (AST in what follows). Each step matches a AST node whose type name is the name of the step. `/` represents the root node, `.` represents the current node, `..` represents the parent of current node, while `//` represents any successor node(s) of the given type.

XPath function calls could be used, represented as `functionName(arg1, ..., argN)`. All steps are relative to current node in context (typically initial context contains just the root of the AST).

Each step could be qualified by an XPath axis `axis::step`, representing a navigation from the current node. The default axis  is children, while `..` is a shortcut for `parent::`, `//` shortcut for `descendant::`, and `@attribute` a shortcut for `attribute::` axis that returns the value of AST node property with that name (result of getter method with that name, for the node).

Some examples make the XPath syntax clearer:

- `//Class/Field[@Name='x']`

matches the AST nodes of type "Field" under any node of type Class, with "x" as the value of the Name property (getName() getter).

- `/Class[1]/Method[not(FormalParameter)]`

matches Method under the first top-level class, without FormalParameter children (i.e. no-parameter methods).

- `//CallStatement[ matches(@Name, 'PATTERN') ]/following-sibling::*[1]`

matches the next AST node following any call to a function matching PATTERN.

The following figure shows which nodes correspond to the different XPath axes available (excluding attribute axis):

[blocked URL](#)

XPath functions reference

NOTE: In the function signatures, optional arguments are enclosed between [and]. A sequence of multiple arguments is represented by '...'.

Basic XPath functions:

Function	Meaning	Example
<code>empty(a1, ..., aN)</code>	True if no args or all args are empty lists or null	<code>//Class[empty(Field, Method)]</code>
<code>qak:every(var, binding, test)</code>	True if test XPath expr (executed on each value in binding expr bound to var) is true for all values in the binding expression.	<code>//ConstructorDeclaration[qak:every('\$call', qak:find('calls', .), 'CHECK_ON_METHOD_CALL')]</code>
<code>except(A, B)</code>	Nodeset difference: Return items in A but not in B (alias: difference())	<code>except(//Field //Method, /*[matches(@Name, 'get')])</code>
<code>exists(a1, ..., aN)</code>	True if args and each arg is non-empty list or non-null	<code>//Class[exists(Field, Method)]</code>
<code>qak:for(var, binding, returnExpr)</code>	Return set with values returned by executing returnExpr for each value in the binding expression, with var bound to value on each execution.	<code>qak:for('\$x', //Method, 'qak:search("container", \$x)')</code>
<code>qak:getCommentOn()</code>	Return the code comment on the context node (null if no comment)	<code>//Method[contains(qak:getCommentOn(), 'password')]</code>
<code>qak:groovy(\$code, args)</code>	Evaluate groovy closure with given args	<code>//ClassOrInterfaceDeclaration[qak:groovy('{args, ctx -> args.isPublic()}', .)]</code>
<code>qak:if(\$test, \$expr1, \$expr2)</code>	Return value of \$expr1 if boolean(\$test) is true, \$expr2 otherwise. If no \$expr2 is provided, empty nodeset is returned.	<code>qak:if(\$clazz[@Interface='true'], \$clazz /Method, \$clazz/Method[@Public='true'])</code>
<code>qak:in(\$left, \$right)</code> \\ <code>qak:in(\$right)</code>	Return true if \$left is contained in \$right (all elements in \$left are in \$right). \$left is the context set if not explicitly given.	<code>qak:search('variableDeclaration', qak:navigate('variableUsages', //VariableDeclaratorId)) [qak:in(//LocalVariableDeclaration)]</code>
<code>intersect(\$first, \$second)</code>	Intersection	<code>intersect(//Field //Method, /*[matches(@Name, 'get')])</code>
<code>matches(arg, regularExpr)</code>	True if any substring for arg matches the regular expression	<code>//Method[matches(@Name, 'get')]</code>
<code>max(a1, ..., aN)</code>	Max value in the input argument	<code>//Method[@BeginLine < max(..Field /@BeginLine)]</code>
<code>min(a1, ..., aN)</code>	Min value in the input argument	<code>//Field[@BeginLine > min(..Method /@BeginLine)]</code>
<code>node-after(\$one, \$two)</code>	True if \$one appears later in code than \$two	<code>//Field[node-after(.. /Method)]</code>
<code>node-before(\$one, \$two)</code>	True if \$one appears before in code than \$two	<code>//Method[node-before(.. /Field[last()])]</code>
<code>replace(\$str, \$regexp, \$subst)</code>	Replace groups matched in str by regexp with subst pattern	<code>replace(@Name, '^(is get has)(.+)\$', '\$2')</code>

reverse(\$nodeset)	Reverse items in nodeset	reverse(//Statement)
root() / root(\$arg)	Get root node	except(//Method, root()/Class[1]/Method)
qak:some(var, binding, test)	True if test expr (executed on each value in binding expr bound to var) is true for at least one value in binding expression.	//ConstructorDeclaration[qak:some('\$call', qak:find('calls', .), 'SOME_CHECK_ON_METHOD_CALL')]
subsequence(\$arr, \$init [, \$size])	Extract subsequence from arr starting from init (starts at 1) up to size	subsequence(//Class, 2, count(//Class - 2))
qak:strict-path (\$nodes, \$p1, ..., \$pN) qak:strict-path (\$p1, ..., \$pN)	Return nodes (in \$nodes or in context nodes when not present) that have degenerate subtree composed of the p1 ... pN successor node types.	qak:variable('\$includes', //IncludeStatement //IncludeExpression) \$includes[not(qak:strict-path ('Expression', 'UnaryExpression', 'PrimaryExpression', 'String'))]
qak:symmetric-difference(\$one, \$two)	Items in one of the sets but not both	qak:symmetric-difference(//Class/Method [@Public='true'], /Class/Method[starts-with(@Name, 'test')])
union(a1, ..., aN)	Same as a1 ... aN	union(//Method, //Constructor)
qak:variable(var, expr)	Sets var to expr, returning empty set (so it could be used in union expressions). Similar to <xsl:variable>	qak:variable('\$f', //Field) qak:variable('\$m', //Method) except(\$m, qak:hla(qak:navigate ('variableUsages', \$f))/ancestor::Method)



For better understanding of XPath functions, you may execute the sample XPath expressions in Rule Developer and try to understand for each example what nodes are looked for.

For example,

```
//ConstructorDeclaration[
  qak:every('$call', qak:find('calls', .), 'matches( $call
/PrimaryPrefix/@Label, "super\."')')
]
```

looks for constructors (in Java) where all calls are to methods in super class.

Extended XPath functions

Extended XPath functions use primitives, like predicates, visitors, navigations or visitors:

Category	Function	Meaning	Example
Search & Navigation	qak:accept (\$visitor, \$arg)	Apply visitor to nodes in \$arg.	qak:accept(\$myVisitor, //MethodDeclaration)
Search & Navigation	qak:filter (\$pred, \$arg)	Return nodeset with nodes in \$arg matching NodePredicate \$pred.	qak:filter('hasUsages', //VariableDeclaratorId)
Search & Navigation	qak:find (\$pred, \$arg)	Return nodeset with all nodes matching pred under each subtree rooted at each node in \$arg	//MethodDeclaration[not(qak:find('calls', .))]
Search & Navigation	qak:navigate (\$navigation, \$arg)	Return nodeset with nodes reachable after running \$navigation starting at each node in \$arg	qak:navigate('container', //PrimaryExpression[...]) qak:navigate('variableUsages', //VariableDeclaratorId)
Search & Navigation	qak:query (\$query, \$arg)	Return nodeset with nodes reachable after running \$query starting at nodes in \$arg	qak:query(\$myQuery, //VariableDeclaratorId) //VariableDeclaratorId[qak:query(\$myQuery)]
Search & Navigation	qak:search (\$search, \$arg)	Return nodeset with nodes found after executing \$search on each node in \$arg.	qak:search('methodDeclaration', qak:find('calls')) qak:find('calls')[qak:search(\$search)]
HLA/LLA conversion	qak:hla() / qak:hla(\$arg)	Return HLA nodes for each context node or each node in \$arg	qak:hla(/xpath/expr/to/nodes, /other/xpath/expr/to/nodes) qak:hla()/path/on/hla
HLA/LLA conversion	qak:lla() / qak:lla(\$arg)	Return LLA nodes for each context node or each node in \$arg	qak:lla(/xpath/expr/to/nodes, /other/xpath/expr/to/nodes)

Language-specific XPath functions

For some technologies, a few additional XPath functions (prefixed with qak-LANGUAGE) are provided.

Function	Meaning	Example
qak-java:typeof (\$val, \$full[, \$short])	True if \$var (an attribute) match full/short type name, or node matches type	//ClassOrInterfaceDeclaration[qak-java:typeof(@Image, 'java.io.Serializable')]

XPath API

Often you may use the XPath API to build an XPath expression that you may execute in a rule: Part of a complex search could be represented succinctly by an XPath expression, while other parts of the query could be implemented by other means. For that, XPath class provides an interface to declarative queries in the AST.

The `com.als.core.xpath.XPath` class (in *qakingCore* module) represents an XPath expression that could be executed from a start node (typically a `BaseNode`). The XPath interface is rather simple:

```
XPath(String xpath) throws JaxenException;
XPath(String xpath, RuleContext ctx) throws JaxenException;

List selectNodes(Object initial);
Object selectSingleNode(Object initial);
Object evaluate(Object initial);
```

Sample usage that could be used in custom rules:

```
BaseNode astNode = ...;
XPath xpath = new XPath("//MyNode/SubNode[@Image='x']", ruleContext);
List nodes = xpath.selectNodes(astNode); // A list of BaseNode (possibly
empty)
BaseNode node = xpath.selectNode(astNode); // The first node matching the
XPath expr, or null

XPath xpath2 = new XPath("//MyNode/SubNode/@Image");
List images = xpath.evaluate(astNode);
```

The `astNode` passed could be high-level or low-level AST, depending on the node that you pass as the initial node.

NOTE: Other parts of the API accept an XPath expression argument. For example, `com.optimyth.qaking.highevelapi.navigation.XPathRegion` builds a navigation based on XPath expression. And in `com.optimyth.qaking.query.Query` some methods have an XPath expression as argument:

```
// Visit nodes reachable from current context by XPath expression
visit(XPath xpath, NodeVisitor visitor);
visit(String xpath, NodeVisitor visitor);

// Navigate to nodes reachable by XPath expression, starting from current
query context
navigate(XPath xpath);
navigate(XPath xpath, NodePredicate pred);
navigate(String xpath);
navigate(String xpath, NodePredicate pred);
```

Extending XPathRule

Usually an XPath custom rule does not need a custom Java class with explicit logic. You simply specify the Java class `com.als.core.rule.XPathRule` in the descriptor, and provide the XPath expression in the "xpath" rule property.

In some cases, you may need to set XPath variables to the XPath expression, or provide a non-default logic for reporting violations at nodes returned by the XPath expression. `XpAthRule` could be extended, by overwriting two methods:

```

/**
 * Invoked in rule initialization. Subclasses may add additional logic for
 * configuring the XPath statement,
 * like setting XPath variables. Default implementation does nothing.
 */
protected void initializeQuery(XPath xpath, RuleContext ctx) {}

/**
 * Invoked on each BaseNode matched by the XPath query. Subclasses may
 * change the rule violation reporting logic.
 * Default implementation simply create and report a violation on the
 * node begin line.
 * @param node BaseNode where the violation will be reported.
 * @param ctx RuleContext
 * @return RuleViolation created (added to the report).
 */
protected RuleViolation reportViolation(BaseNode node, RuleContext ctx) {
    int line = node.getBeginLine();
    if(line<=0) line = TreeNode.on(node).findLine();
    RuleViolation rv = createRuleViolation(ctx, line);
    ctx.getReport().addRuleViolation( rv );
    return rv;
}

```