PsychoPath

PSYCHOPATH

http://psychopath.sourceforge.net

Presentation Outline

- XML and XPath terminology.
- Basic XPath 2.0 Queries.
- Key Requirements of PsychoPath.
- PsychoPath project details:
  - Design and implementation.
  - Testing.
  - Performance.
  - Evaluation.
- Conclusions and Future Work.

Introduction

- XML is a mark-up language that defines objects and the relationships between them in documents.
- XPath is a language to address, extract and view particular parts of an XML document.
- XPath 2.0 is a language update that introduces XML Schema awareness and simple/complex types.

- PsychoPath is a Schema Aware XPath 2.0 processor.
- Main competitor is Saxon written by Michael Kay – one of the authors of the XPath 2.0 specification.
    - Open source version of Saxon does not support XML Schema.

Basic XPath 2.0 Queries

```
<shop>

   <item>
      <name>Flour</name>
      <price>10.01</price>
   </item>

   <item>
      <name>Cake</name>
      <price>10</price>
   </item>

   <item>
      <name>Egg</name>
      <price>10.0</price>
   </item>

</shop>
```

Example 1

//item[2]

returns the second item node

Example 2

//item[./price=10]

returns the *Cake* and *Egg* item nodes

XPath 2.0 specification: http://www.w3.org/TR/xpath20/

Key Requirements

- Good Object-Oriented architecture.
- Modularity.
- Components fully tested.
- Components and Test suites easily extendable.
- Full implementation of the XPath 2.0 grammar.

- Implementation of as many XPath 2.0 types, operators and functions as time permits.
- Analyse performance considerations only if time remains and not at expense of above requirements.

The Problem

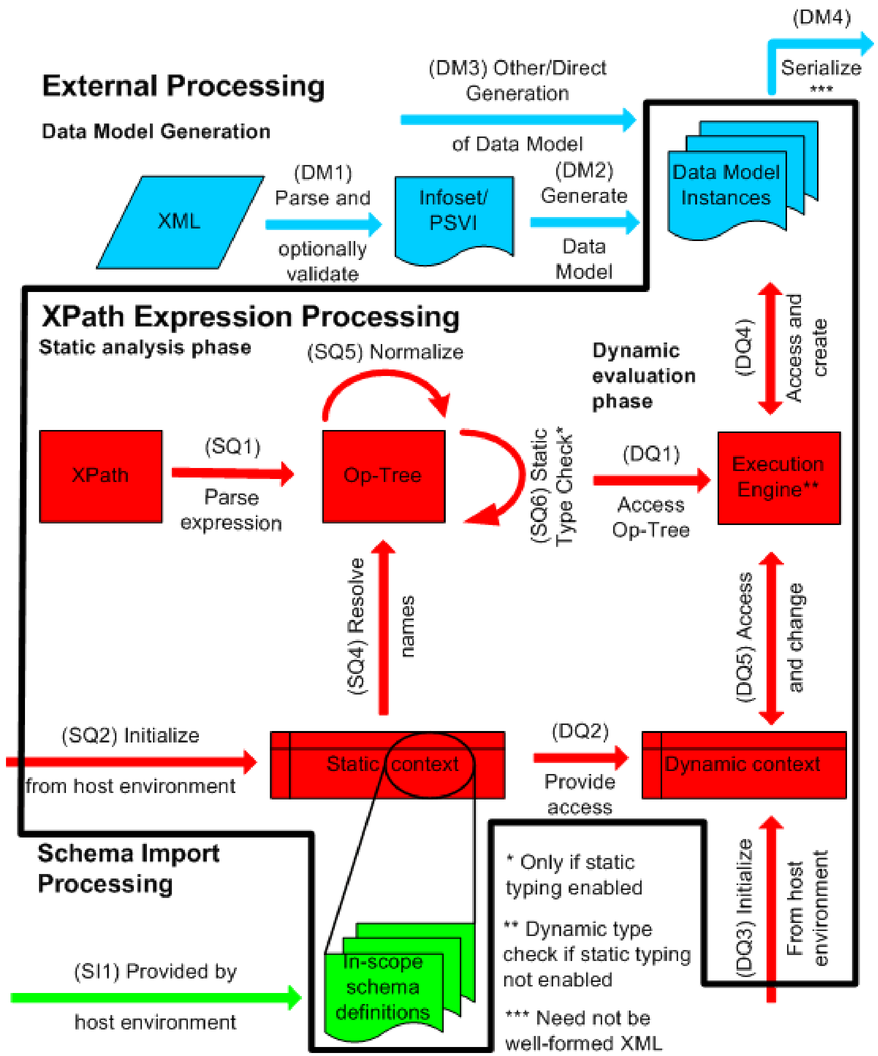


Diagram taken from the XPath 2.0 specification

Development Process

Four main iterations:

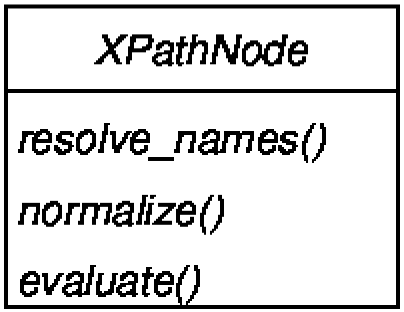
1. XPath parsing and DOM loading.
2. Static Analysis.
3. Dynamic Analysis.
4. Implement more functions, types and operators.

Parser – JFlex & CUP

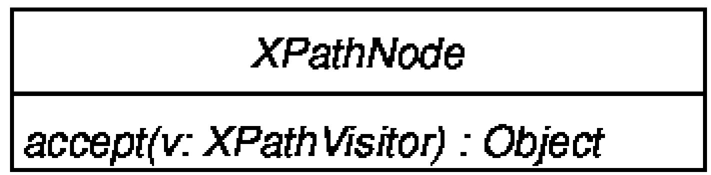
- ✔ Less time spent on writing the parser.
- ✔ Isolates parsing the code from the grammar.
- ✔ Easier to maintain and debug.

- ✗ Runtime inefficiency.

Abstract Syntax Tree (AST)



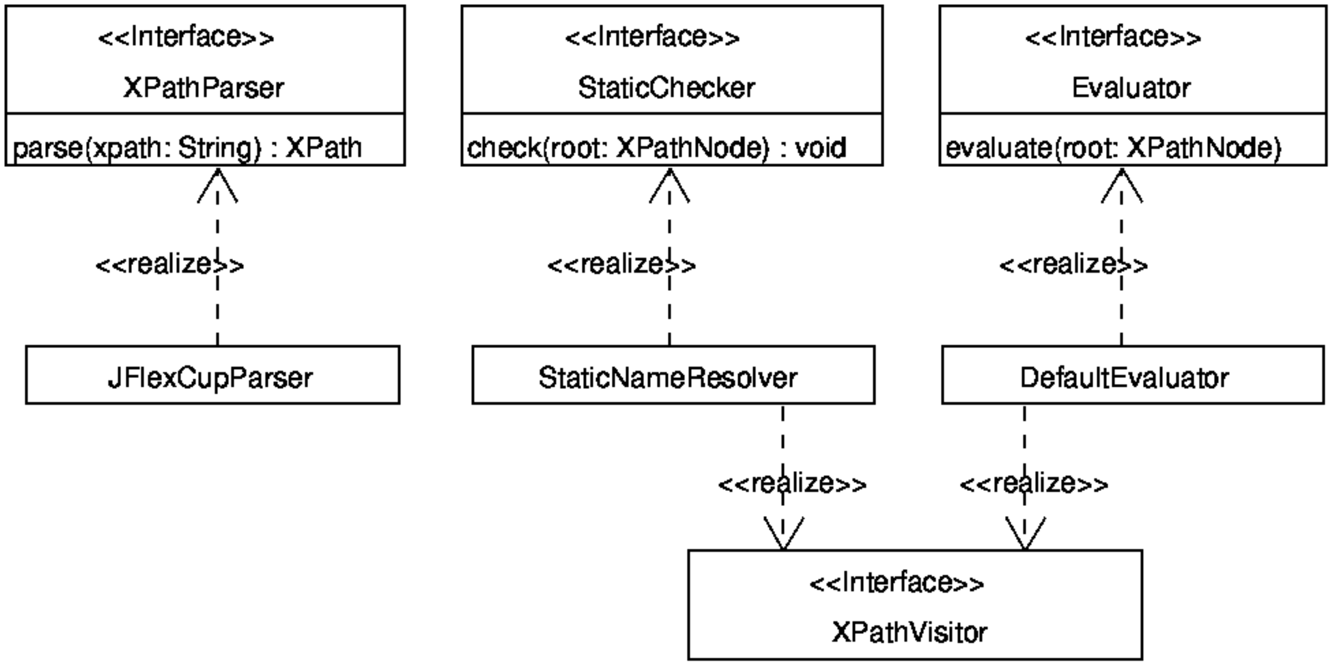
- ✔ Efficient runtime.
- ✗ Need to modify all AST in order to add a new operation.



- ✔ No changes to AST required for a new operation.
- ✔ Multiple implementations per operation possible.
- ✔ Code for operation is localized and not spread throughout AST.

- ✗ Runtime inefficiency – Double dispatch.

Modularity



- Execution phases are independent.
- Top level interfaces are not tied to the Visitor Pattern.

XPath Functions

Over 100 functions are defined in XPath 2.0.

- Implementation per function must be minimal.

Function
<<create>> Function(name: QName,arity: int) *evaluate(args: Collection) : ResultSequence*

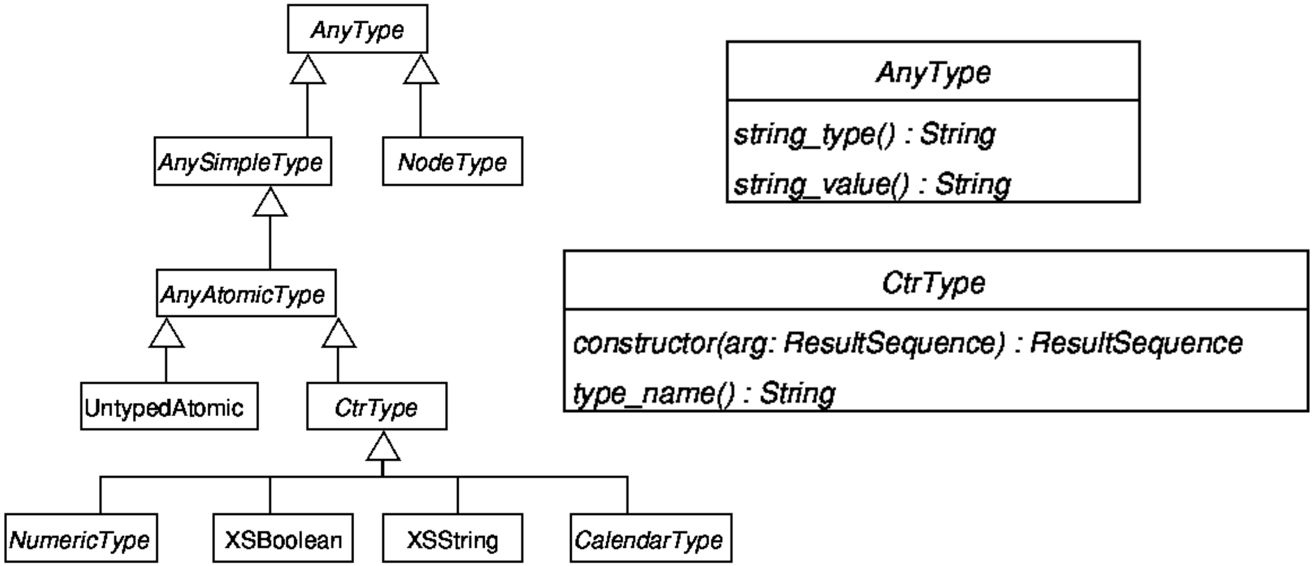
- Function *signature* is defined by its name and arity.
- Each function is registered with a *library*.

Function dispatch:

- Function signature is used as a lookup key in the library.
- The entry will be the function to evaluate.

XPath Types

Support for XML Schema types (over 25 defined).



- Constructor function automatically defined on types which derive from *CtrType*.
- Operators are implemented by the types.

User Interface

Results are returned in a *ResultSequence*.

- List of items of type *AnyType*.
- Need to identify concrete type of object.
    - User frequently knows the type.

Example for the XPath expression “//*”:

Load XML	DOMLoader loader = new XercesLoader(); Document doc = loader.load(xml); DynamicContext dc = new DefaultDynamicContext(null, doc);
Parse XPath	XPathParser xpp = new JFlexCupParser(); XPath root = xpp.parse(“//*”);
Static Analysis	StaticChecker name_check = new StaticNameResolver(dc); name_check.check(root);
Dynamic Analysis	Evaluator eval = new DefaultEvaluator(dc, doc); ResultSequence rs = eval.evaluate(root);

Useful work	for(Iterator i = rs.iterator(); i.hasNext();) {
	    NodeType node = (NodeType) i.next();
	    do_dom_node(node.node_value());
	}

Testing

Fine grained testing of each class:

- ✔ Greater confidence that all code is tested.
- ✔ Precise knowledge of what caused a test to fail.
- ✗ Less confidence that the code works as a *whole*.
- ✗ Large overhead in maintaining tests – classes change frequently.

Testing of main interfaces:

- ✔ High level interfaces change less frequently.
- ✔ Confidence that components interact properly.
- ✔ Less test *code* (not test *cases*).
- ✗ Hard to ensure test coverage.
- ✗ Hard to ensure that expected failures occurred for a specific reason.

Test Cases

- Test cases (input, answer pair) are held in an external text file.
  - ✔ Easy to add test cases.
  - ✔ No need to recompile suite on a new test.

Example:

```
1+1
****
1) xs:integer: 2

****
1/0
****
% div by 0!
FAIL
****
```

Statistics:

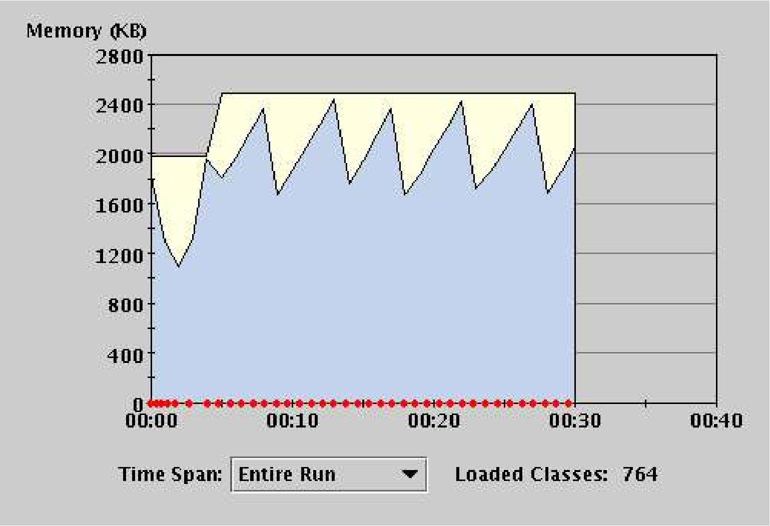
- 100% Test coverage in main packages according to JProbe.
- Over 900 test cases.

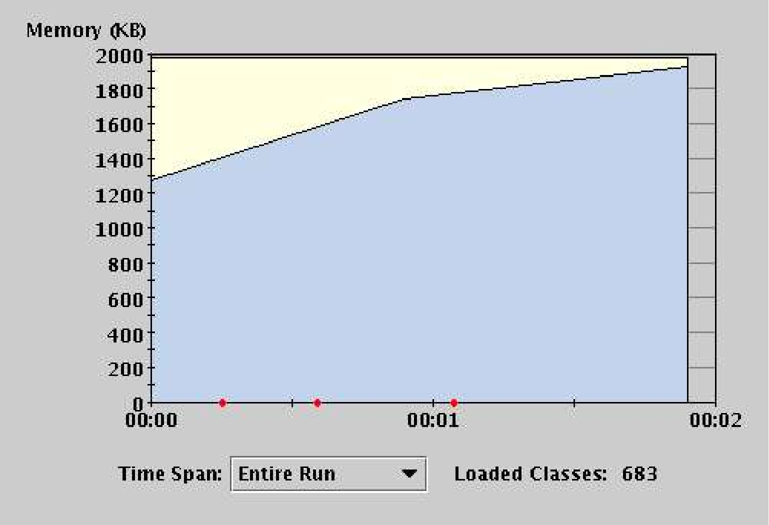
Performance

Memory usage on evaluation of the XPath expression:

(10 to 20000)[19909]

On milestone 4, it took 400ms to evaluate.





Current status.

- *ResultSequence* objects created via a factory.
- Lazy evaluation of range expressions.

Constant evaluation time of 30ms.

Demo

4 node Travelling Salesman Problem

```
tokenize( ( for $nodec in count(distinct-values(for $j in /
graph/edge/@src return string($j))) return( for $cost in (0
to xs:integer(tokenize(string(max(for $i in /graph/*/@cost
return xs:integer(string($i))) * xs:double($nodec)),'\.')[1])),
$path in  (for $A in (/graph/edge[@src = 'A']) return for $B
in (/graph/edge[@src = string($A/@dst)]) return for $C in
(/graph/edge[@src = string($B/@dst)]) return for $D in /
graph/edge[@src = string($C/@dst)][@dst = 'A'] return
concat( string(xs:integer(string($A/@cost)) + xs:integer
(string($B/@cost)) + xs:integer(string($C/@cost)) +
xs:integer(string($D/@cost)) ), '|' , string($A/@src), string
($B/@src), string($C/@src), string($D/@src), 'A'  ) ) return
( if( (count(distinct-values(tokenize( substring-after
(tokenize($path, '\|')[2], 'A')    , '()'))) = ($nodec + 1)) and
starts-with($path, concat( string($cost), '|'))) then $path
else () ) )[1] ), '\|')
```

Evaluation

- PsychoPath
    - Substantial amount of the specification has been implemented.
    - Extendable design allows for further implementation.
    - Not all XPath types implemented.

- PsychoPath vs. Saxon:
    - Saxon is faster – heavily optimised.
    - Saxon is backwards compatible with XPath 1.0.
    - *Commercial* version of Saxon is schema aware.
    - PsychoPath is able to handle some expressions better.
    - PsychoPath is schema aware and open-source.

Conclusion

- We met our goals and created the first free schema aware XPath 2.0 processor, which supports about 75% of the specification.
- PsychoPath is fully usable and is a good competitive product against Saxon. Although not fully optimized presents a real useful alternative with its schema awareness.
- Communication and flexibility was the key to success.
- There is a solid base for future work:
  - re-factoring, optimisation, implementing the full specification.

Where is our money?

COCOMO

- Total Physical Source Lines of Code = 17,622
- Development Effort Estimate, Person-Years = 4.07 (48.82 months)
- Schedule Estimate, Years = 0.91 (10.95 months)
- Estimated Average Number of Developers (Effort/Schedule) = 4.46
- Total Estimated Cost to Develop = $ 549,540 (average salary = $56,286/year, overhead = 2.40)