Identifying Query Incompatibilities with Evolving XML Schemas

Pierre Genevès

(with Nabil Layaïda and Vincent Quint)
CNRS and INRIA

The ACM International Conference on Functional Programming
ICFP'09 – Edinburgh, UK – September 1st, 2009

If XML is the solution, then what was the problem?

If XML is the solution, then what was the problem?

Initial Goal

- Ensuring long-term access to data (documents)
- Write documents in 1998 and read them safely in 2087

→ Build an external grammar and encode your data in XML!
Advantages:

  - Markup language for describing (structured) data in itself (independently from processors)
  - No need to write parsers anymore (widely available)

If XML is the solution, then what was the problem?

Actual use of XML

- Web standards: HTML 1, 2, 3.2, XHTML 1.0, XHTML 1.1 (Second Edition), Microsoft IE's HTML, Mozilla's SVG, SMIL patched by Nokia ...

  "The nice thing about standards is that there are so many of them"

- Application-specific schemas: my schema beta version 0.1, Fred's version patched with Mike's hack, yesterday's version with tomorrow extensions...
- Applications using XML rapidly evolve today

$\rightarrow$ The initial problem "How to ensure long-term access to data?" now becomes: "**How to deal with frequent grammar evolutions and their impacts on programs evolution?**"

Today's Typical XML Application

- An XML program takes as input both:
  - XML instances (tree structures)

```
<Vita>
   <Born><When>August 22, 1862 </When><Where>Paris</Where></Born>
   <Married><When>October 1899</When><Whom>Rosalie</Whom></Married>
   <Married><When>October 1899</When><Whom>Rosalie</Whom></Married>
   <Died><When></When><Where>Paris</Where></Died>
</Vita>
```

  - XML types (tree grammars defining constraints on children and siblings of nodes using regular expressions)

```
<!ELEMENT Vita (Born, Married*, Died?)>
<!ELEMENT Born (When, Where)>
<!ELEMENT Married (When, Whom)>
<!ELEMENT Died (When, Where)>
```

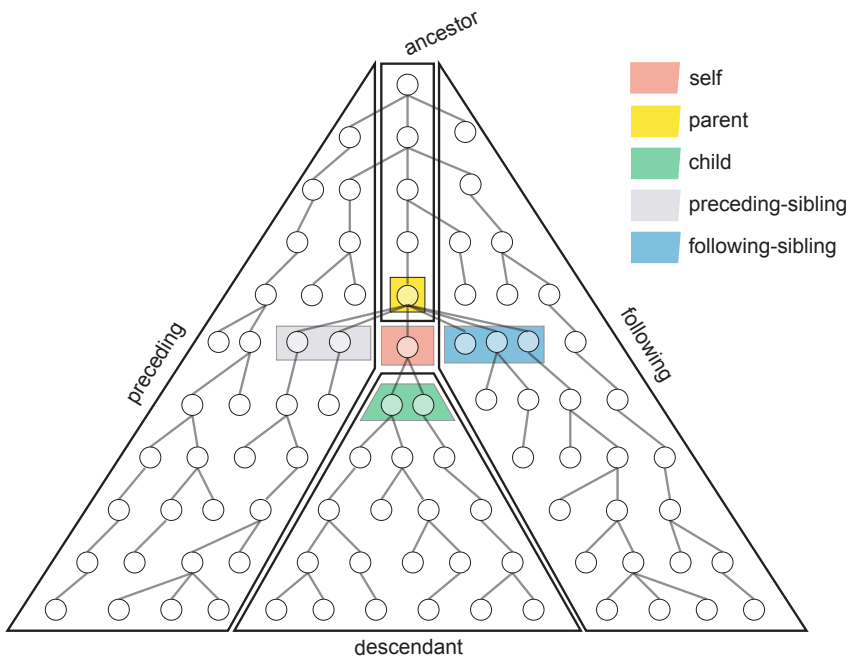
- It usually performs 2 essential tasks before writing some output:
  - Validation: check that an XML document is valid w.r.t. a given type
  - Navigation/Extraction: select parts of a document to be transformed (XPath expressions)

XPath Expressions

- XPath: standard query language for navigating and extracting information from XML trees
- XPath expressions return a set of matching nodes
- Vertical regular expressions $(axis::nodetest[filter]'/')^{n}$

Example

parent::company/descendant::staff[not parent::manager]



The Fundamental Problem

- Program P processes documents of type T.
- Type T evolves into $T' = T \oplus \Delta$.
- Can I still use P safely with documents of type T' ?

Yes

- We want nothing but the proof for all instances of T' (static type checking)

No: P requires an update

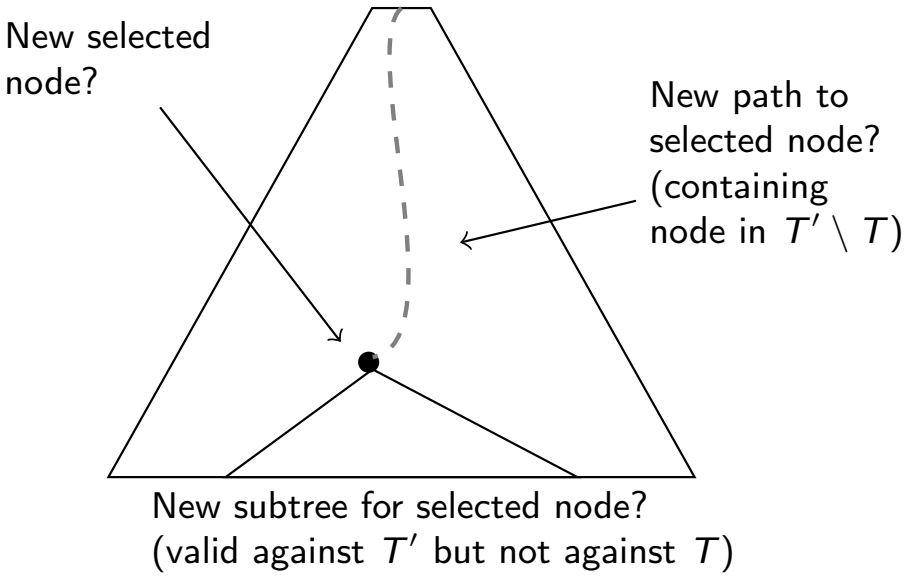
- We want more than the proof (counterexample): can we know
  - how Δ affects P?
  - which parts require to be updated?
  - for which reasons?

$\rightarrow$ Need for tools allowing to diagnose and fix problems due to evolution

1. How does T' relate to T (or differ from T)?
2. What about XPath queries in P?

Is a Query Concerned by Type Evolutions?

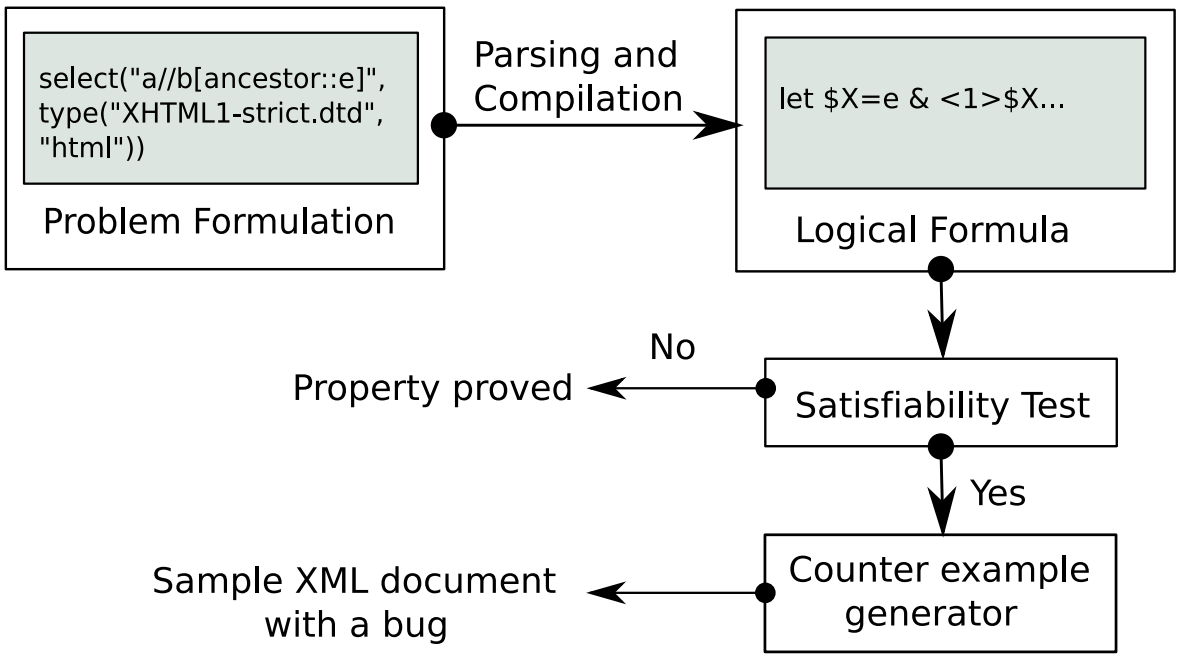
Several possible scenarii (and combinations)



What makes automated identification complex?

- Time complexity of reasoning with grammars (regular tree languages) and XPath (multi-directional recursive navigation) is at least in EXPTIME.

Proposed System Architecture

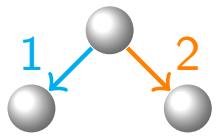


Advantages

- Predicate language with XPath and common XML schemas syntax
- Problem can be formulated in a unifying logic that capture evolution problems
- Appropriate logic is decidable in $2^{O(n)}$ [Geneves et al., PLDI'07]
- Provide formal proofs of compatibility or detailed counterexamples

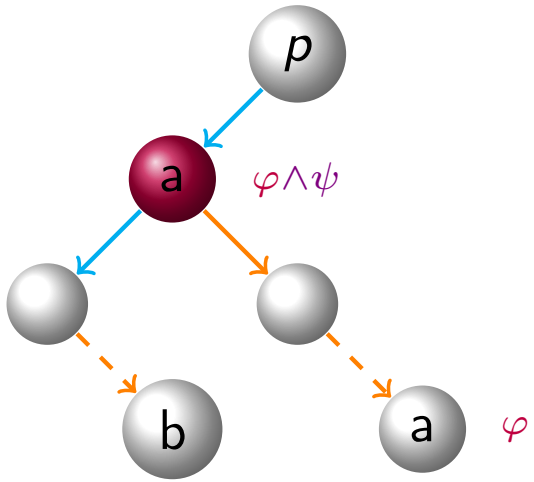
Zoom on Logical Formulas: the "Assembly Language"

- XML trees are seen as labeled binary trees (wlog)
- Programs $\alpha \in \{\mathsf{fc}, \mathsf{ns}, \overline{\mathsf{fc}}, \overline{\mathsf{ns}}\}$ for navigating binary trees ($\overline{\overline{\alpha}} = \alpha$)



φ, ψ	$::=$		formula
		$\top$	true
	$\mid$	$\sigma \quad \mid \quad \neg\sigma$	atomic proposition (negated)
	$\mid$	$\varphi \vee \psi \quad \mid \quad \varphi \wedge \psi$	disjunction (conjunction)
	$\mid$	$\langle\alpha\rangle\,\varphi \quad \mid \quad \neg\langle\alpha\rangle\,\top$	existential (negated)
	$\mid$	$\mu X.\varphi$	unary fixpoint (finite recursion)
	$\mid$	$\mu\overline{X_i.\varphi_i}$ in ψ	n-ary fixpoint

Translating XPath and Types in the Logic



- Formula holds at selected nodes
- $\mu Z.\varphi$: finite recursion
- Converse programs are crucial
- Types can be translated as well
- It is easy to "tag" (associate an atomic proposition to) any set of nodes

Translated query: $p/\text{child}::a\,[\text{child}::b]$

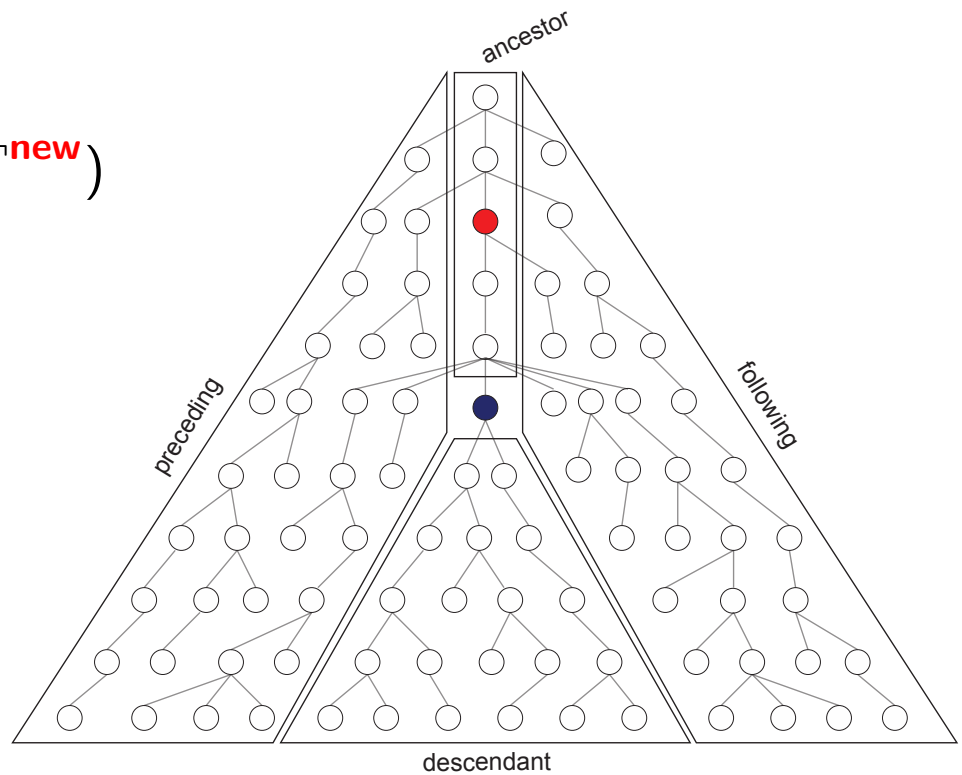
$$\underbrace{a \wedge (\mu Z.\, \langle \overline{\text{fc}} \rangle\, p \vee \langle \overline{\text{ns}} \rangle\, Z)}_{\varphi} \quad \wedge \underbrace{\langle \text{fc} \rangle\, \mu Y.\, b \vee \langle \text{ns} \rangle\, Y}_{\psi}$$

Introducing Predicates

- Predicates that capture type differences are logically definable!
- Key ideas:
  - Tag type nodes to distinguish them in problem formulation
  - Formulate a bad scenario and check it for satisfiability
  - Rely on XPath's partitioning of tree nodes

- Example: May Q select T nodes in new contexts with T'?

$$\texttt{new\_region}(\texttt{"}Q\texttt{"}, T, T') \stackrel{\text{def}}{=}$$
$$\texttt{select}(\texttt{"}Q\texttt{"}, \text{trans}(\text{T'}) \wedge \text{trans}(\neg T)^{\neg \textbf{new}})$$
$$\wedge \neg\texttt{added\_element}(T, T')$$
$$\wedge\, \texttt{ancestor}(\textbf{new})$$
$$\wedge\, \neg\texttt{descendant}(\textbf{new})$$
$$\wedge\, \neg\texttt{following}(\textbf{new})$$
$$\wedge\, \neg\texttt{preceding}(\textbf{new})$$



A Predicate Toolbox

$$
\begin{aligned}
\textit{corePredicate} \quad ::= \quad & \texttt{select}(\texttt{"}\textit{query}\texttt{"}, \varphi) \\
\mid \quad & \texttt{exists}(\texttt{"}\textit{query}\texttt{"}, \varphi) \\
\mid \quad & \texttt{type}(\texttt{"}f\texttt{"}, l) \\
\mid \quad & \texttt{forward\_incompatible}(\varphi, \varphi') \\
\mid \quad & \texttt{backward\_incompatible}(\varphi, \varphi') \\
\mid \quad & \texttt{element}(\varphi) \\
\mid \quad & \texttt{attribute}(\varphi) \\
\mid \quad & \texttt{descendant}(\varphi) \\
\mid \quad & \texttt{exclude}(\varphi) \\
\mid \quad & \texttt{added\_element}(\varphi, \varphi') \\
\mid \quad & \texttt{added\_attribute}(\varphi, \varphi') \\
\mid \quad & \texttt{new\_element\_name}(\texttt{"}\textit{query}\texttt{"}, \texttt{"}f\texttt{"}, \texttt{"}f'\texttt{"}, l) \\
\mid \quad & \texttt{new\_region}(\texttt{"}\textit{query}\texttt{"}, \texttt{"}f\texttt{"}, \texttt{"}f'\texttt{"}, l) \\
\mid \quad & \texttt{new\_content}(\texttt{"}\textit{query}\texttt{"}, \texttt{"}f\texttt{"}, \texttt{"}f'\texttt{"}, l)
\end{aligned}
$$

Example

- We take a program P (which is an XSLT transform from MathML 1.0 Structure to Presentation)
- We want to know what happens when P is fed with MathML 2.0 documents
- The process:
  1. Extract Q_i from P

```
Q1:  //apply[*[1][self::eq]]
Q2:  //apply[*[1][self::apply]/inverse]
Q3:  //sin[preceding-sibling::*[position()=last()
                and (self::compose or self::inverse)]]
      ...
```

  2. Check each Q_i for potential incompatibilies

Example: MathML Evolution

- Does Q1 select nodes in new contexts in MathML 2.0?

```
new_region("Q1","mathml.dtd","mathml2.dtd","math")
```

```
<math xmlns:solver="http://wam.inrialpes.fr/xml"
            solver:context="true">
  <declare>
    <apply solver:target="true">
      <eq/>
    </apply>
    <condition/>
  </declare>
</math>
```

1. Yes: `Q1` selects "`apply`" elements whose ancestors can be "`declare`" elements which was not possible with v1.0
2. Worst, this evolution breaks type-safety of P since $P(\textit{counterexample})$ does not validate against MathML 2.0

Example: MathML Evolution

- Does Q2 select nodes with new subtrees regardless of new elements?

```
new_content("Q2","mathml.dtd","mathml2.dtd","math")
& exclude(added_element(type("mathml.dtd","math"),
                        type("mathml2.dtd", "math")))
```

```
<math xmlns:solver="http://wam.inrialpes.fr/xml"
            solver:context="true">
  <apply solver:target="true">
    <apply>
      <inverse/>
    </apply>
    <annotation-xml>
      <math/>
    </annotation-xml>
    <condition/>
  </apply>
</math>
```

- The counterexample effectively exhibits a new combination of MathML 1.0 elements in MathML 2.0.

→ Evolutions of standards may break existing programs (software cannot simply ignore new elements)

Concluding Remarks

- Evolution of standards should be more formally checked...
  - Some of the most widely used standards are not forward/backward compatible (see paper)
- Logical frameworks can be successfully used to identify undesired effects of evolution on programs involving complex constructions
- The tool helps assessing the amount of changes required to follow type evolution
- Web application at:

http://wam.inrialpes.fr/xml

Appendix

XHTML Basic Example (1/2)

```
backward_incompatible("xhtml-basic10.dtd",
                      "xhtml-basic11.dtd", "html")
```

- Immediate counterexample as new schema contains new element names:

```
<html>
  <head>
    <title/>
    <style type="_otherV"/>
  </head>
  <body/>
</html>
```

- "`style`" element can now occur as a child of `head` (which was not permitted in XHTML basic 1.0)

XHTML Basic Example (2/2)

```
backward_incompatible("xhtml-basic10.dtd",
                      "xhtml-basic11.dtd", "html")
& exclude(added_element(
      type("xhtml-basic10.dtd","html"),
      type("xhtml-basic11.dtd", "html")))
```

```
<html>
  <head>
    <object>
      <label>
        <a>
          <img/>
        </a>
        <img/>
      </label>
      <param/>
    </object>
   ...
```

- XHTML basic 1.1 is not backward compatible with XHTML basic 1.0 even if new elements are not considered
- The "label" element cannot have an "a" element with v1.0 (while it can with v1.1)
- Similar incompatibilities with SVG, SMIL,... W3C standards

→ Applications cannot simply ignore new elements from newer schemas!

A Few Remarks....

- The most complex of the previous tests were processed in less than 30 seconds on an ordinary laptop computer running Mac OS X.
- Most complex: analyzing recursive forward/backward and qualified queries (such as `Q3`), under evolution of large and heavily recursive schemas such as XHTML and MathML (large number of type variables, elements and attributes)
- For the most complex tests, problem formula size is 550. The search space explored by the solver is $2^{550} \approx 10^{165}$... more than the square number of atoms in the universe 10^{80}
- More from:

http://wam.inrialpes.fr/xml