HAL
archives-ouvertes.fr

Equipping IDEs with XML-Path Reasoning Capabilities

Pierre Genevès, Nabil Layaïda

▶ To cite this version:

Pierre Genevès, Nabil Layaïda. Equipping IDEs with XML-Path Reasoning Capabilities. ACM Transactions on Internet Technology, Association for Computing Machinery, 2014, 13 (4), pp.20. <10.1145/2602573>. <hal-00868723v2>

HAL Id: hal-00868723
https://hal.inria.fr/hal-00868723v2

Submitted on 14 Jan 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A

Equipping IDEs with XML Path Reasoning Capabilities

Pierre Genevès, CNRS
Nabil Layaïda, Inria

One of the challenges in web development is to help achieving a good level of quality in terms of code size and runtime performance, for popular domain-specific languages such as XQuery, XSLT, and XML Schema. We present the first IDE augmented with static detection of inconsistent XPath expressions that assists the programmer for simplifying the development and debugging of any application involving XPath expressions. The tool is based on newly developed formal verification techniques based on expressive modal logics, which are now mature enough to be introduced in the process of software development. We further develop this idea in the context of XQuery for which we introduce an analysis for identifying and eliminating dead code automatically. This proof of concept aims at illustrating the benefits of equipping modern IDEs with reasoning capabilities.

1. INTRODUCTION

Since its introduction, XML has become a de-facto standard for the representation and exchange of structured data and documents. Several programming techniques have been proposed for processing XML data using mainstream general purpose programming languages, and a consensus has emerged around higher level domain-specific languages. In particular XPath [Clark and DeRose 1999; Berglund et al. 2006] is the standard compact notation for extracting information from an XML document. XQuery [Boag et al. 2007], which adds the possibility of constructing new XML documents, has gained considerable interest. This is because an XQuery program can not only easily search, find, and extract relevant information from XML documents using XPath, but it can also perform all sorts of processing, and generate output XML documents, like e.g., web pages. Due to the very popular nature of XML, more and more web applications rely on XQuery code on the server side. The direct usage of XQuery inside web browsers has even been investigated [Fourny et al. 2008]. We believe that code analysis tools for XQuery will be instrumental in the design of modern information systems.

Dead code corresponds to parts of the source code of a program which are executed but whose results are never used in any other computation or yield no result. Presence of dead code reduces runtime performance since computation time is unnecessarily wasted. A particularity of XQuery code is that XQuery programs are very commonly written against a given XML schema [Fallside and Walmsley 2004] that defines constraints with which the queried documents must comply. This provides a dead-code prone setting, as XPath expressions may contain navigational information that contradict requirements expressed by the schema. In that case, the result of the XPath expression is always empty (no matter what the input data are), and all XQuery instructions that depend on this sub-expression are dead code.

Furthermore, in the standardized but ever-evolving context of the web, XML schemas often change as new needs require new features to be added, or new sets of constraints (also known as “profiles”) to be developed. This provides even more favorable circumstances for the existence of dead code (but regrettable circumstances for the XQuery programmer), as it is naturally tempting to share and reuse queries written against similar schema variants, or successive schema versions. For instance, an XQuery program written against the XHTML 1.0 DTD may contain dead code when executed over documents valid with respect to a similar but restricted schema variant

This work was supported by the ANR project TYPEX, ANR-11-BS02-007.

like, e.g., XHTML Basic, or even with documents valid with a newer specification like, e.g., XHTML 2.0 documents.

We propose a system for performing automatic dead-code analysis and elimination from an XQuery program. Furthermore, our system also performs automatic simplification of XQuery code, whenever this is possible, by removing automatically redundancies from XPath expressions. Refactoring XQuery code automatically in those manners offers two benefits: first, it shrinks program size, which helps tracking errors and which is a general preoccupation from a software engineering perspective; and second, it lets the running program avoid executing irrelevant operations, which reduces its running time.

Contribution

The contribution of this article is threefold:

(1) new technical contributions in the static analysis and code refactoring for core XML technologies: the static detection of XQuery dead code with the automatic elimination/refactoring of the corresponding code (detailed as rewriting rules in Section 6); and the elimination of redundancies for the automatic simplification of XPath expressions (detailed as formal rules in Section 4);
(2) the combination of the previous developments with all the required techniques and tools in order to assemble the first IDE equipped with XPath reasoning capabilities;
(3) a novel prototype that provides a proof-of-concept which consists in the integration inside the Eclipse XQDT [XQDT 2011] environment.

Early parts of this work were demonstrated at the International Conference on Software Engineering (ICSE) in 2010 and 2011 [Genevès and Layaïda 2010; Genevès and Layaïda 2011]. The system proposed in the present article requires the resolution of advanced static analysis problems involving XPath expressions, a task for which we build on our earlier works and reuse the logical satisfiability solver presented in [Genevès et al. 2007].

Outline

We first introduce the XML schemas we consider in Section 2, as well as the essential role played by XPath expressions in programs that manipulate XML data in Section 3. In Section 4, we discuss the impact of determining the inconsistency of a given XPath expression. We then present the first augmented IDE with static detection of inconsistent XPath expressions as a proof of concept in Section 5 (see the online video demo [Genevès and Layaïda 2011]). We introduce XQuery programs in Section 6.1, focusing on their particularities that distinguish them from other programs, from a software engineering point of view. We present our dead code analysis for XQuery in Section 6. Finally, we discuss related work in Section 7 before concluding in Section 8.

2. XML DOCUMENTS AND SCHEMAS

XML documents are considered as trees of element and attribute nodes (c.f. Figure 1). A *schema* defines constraints that a particular set of documents should verify (as, e.g., XHTML for web pages). A schema defines the set of admissible elements and attributes in a XML document, as well as how they can be assembled together. This definition is usually done with regular expressions. For example, a simplistic schema for a bookstore (using DTD notation) follows:

```
<!ELEMENT bookstore (book*)>
<!ELEMENT book (title, year, author+)
```

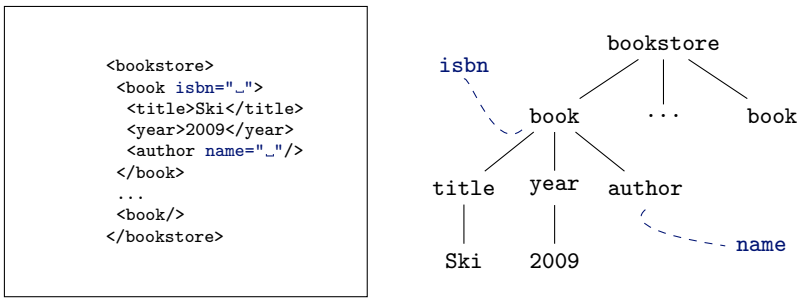

Fig. 1. Sample XML Document and Tree View.

```
<!ATTLIST book isbn CDATA #REQUIRED>
 ...
```

This schema states that a `bookstore` element has any number of `book` elements as children. In turn, each `book` element must have a `title` child, followed by a `year` element and one or more `author` elements. Finally, each `book` element must carry an `isbn` attribute. The sample document shown in Figure 1 is valid with respect to this (partial) schema definition.

2.1. Syntax and Semantics of Schema Languages

There are several notations to express schemas, the most commonly found in practice being DTDs, XML Schemas and Relax NG. Those languages all follow the same paradigm: the content model (the set of admissible children nodes) for a given tree node is described through the use of a regular expression. In fact each schema language corresponds to a particular kind of a tree grammars. For defining more precisely these tree grammars, we start from the syntactic definition of tree type expressions, recalled from [Genevès and Layaïda 2006]. We define a type expression τ as follows:

$$\tau \ ::= \ \emptyset \mid () \mid l[\tau] \mid \tau_1, \tau_2 \mid \tau_1 \mid \tau_2 \mid X \mid \text{let } (X_i \to \tau_i)_{1 \leq i \leq m} \text{ in } \tau$$

where $\emptyset$ is the empty tree type (accepting no tree); $()$ is the type accepting only the empty sequence of trees; $l[\tau]$ is a tree type constructor where l ranges over XML element names and τ defines the content model admissible for the element l; τ_1, τ_2 is the usual concatenation; $\tau_1 \mid \tau_2$ is the usual disjunction (also called "choice" operator in the XML jargon); $X \in \mathit{TVar}$ where TVar is a set of type variables, and "let $(X_i \to \tau_i)_{1 \leq i \leq m}$ in τ" represents the recursive tree type constructor. Notice that with this syntactic representation, the traditional operators found in regular expressions actually correspond to abbreviations: $\tau? = () \mid \tau$, $\quad \tau^+ = \tau, \tau^*$, and $\tau^* = \text{let } X \to \tau \text{ in } \tau, X \mid ()$.

Given an environment θ of type variable bindings, the formal semantics of tree types is given by the denotation function $[\![\cdot]\!]_\theta$:

$$\begin{array}{ll}
[\![\emptyset]\!]_\theta & = \emptyset \\
[\![()]\!]_\theta & = \{()\} \\
[\![X]\!]_\theta & = \theta(X) \\
[\![l[\tau])]\!]_\theta & = \{l'(t) \mid l' \prec l \wedge t \in [\![\tau]\!]_\theta\} \\
[\![\tau_1, \tau_2]\!]_\theta & = \{t_1, t_2 \mid t_1 \in [\![\tau_1]\!]_\theta \wedge t_2 \in [\![\tau_2]\!]_\theta\} \\
[\![\tau_1 \mid \tau_2]\!]_\theta & = [\![\tau_1]\!]_\theta \cup [\![\tau_2]\!]_\theta \\
[\![\text{let } (X_i \to \tau_i)_{1 \leq i \leq m} \text{ in } \tau]\!]_\theta & = [\![\tau]\!]_{\mathit{lfp}(S)}
\end{array}$$

where $\prec$ is a global subtagging relation: a reflexive and transitive relation on labels, and $S(\theta') = \theta[X_i \mapsto [\![\tau_i]\!]_{\theta'}]_{i \geq 1}$.

2.2. The Relation between XML Schema Languages

Each schema language for XML corresponds to a particular set of restrictions over the previously defined tree type expressions. Types as defined above actually correspond to arbitrary context-free tree types, which are much more expressive than XML schema languages found in practice. However, deciding inclusion for such tree types (in the sense of the aforementioned set-theoretics semantics) is known to be undecidable [Hopcroft et al. 2000]. This is why, in practice, restrictions are adopted in order to reduce the expressive power of tree types so that operations like inclusion become decidable or less complex.

Regular Tree Grammars. Regular tree grammars are defined by a syntactic restriction that allows unguarded (i.e. not enclosed by a label) recursive uses of variables, but restricts them to tail positions. For instance the type "let $X \rightarrow (a[()], X) \mid ()$ in X" is allowed, but the type "let $Y \rightarrow (a[()], Y, b[()]) \mid ()$ in Y" is not. This simple syntactic restriction ensures regularity, hence the name of the class. The XML schema languages found in practice correspond to subclasses of regular tree grammars:

DTDs. DTDs correspond to local tree grammars. Local tree grammars are defined as regular tree grammar with the additional restriction that, for each $l[\tau_1]$ and $l[\tau_2]$ occurring in the grammar, the content models are identical: $\tau_1 = \tau_2$. In other terms, it means that the content model for a single XML element is always the same no matter where it occurs in the tree. The content model of an element cannot depend on the context of the element in the tree. This is a very strong restriction which is not always adapted for document modeling.

XML Schemas. XML Schemas correspond to single-type tree grammars. A single-type tree grammar is a regular tree grammar in which, for each $l[\tau_1]$ and $l[\tau_2]$ occurring under the same parent element in the grammar, the content models are identical: $\tau_1 = \tau_2$. This restriction is slightly more permissive than the restriction for DTDs. In XML schemas, the content model for a given element l may depend on the name of any ancestor of l, which is not possible in a DTD, and allows one to describe more flexible content models.

Relax NG schemas. Relax NG schemas directly correspond to regular tree grammars (without any restriction). Regular tree grammars offer even more expressivity when describing tree constraints as the content model of an element may also depend on ancestor's siblings for instance, which is not possible with an XML schema. By definition, regular tree grammars obviously subsume local and single-type tree grammars. For this reason, in this work we consider the class of regular tree grammars, as defined above, as our schema language for XML.

3. PATHS IN XML PROGRAMMING

XPath is the W3C standard language for expressing traversal and navigation in XML data and documents seen as trees. An XPath expression is a succession of navigation steps separated by "/", where each step is made of an axis and a node test. The axis specifies in what direction to search in the tree for nodes matching the node test that can be a label. The last navigation step performs the selection of nodes that form the result of the evaluation of the entire expression. At each navigation step, nodes can be filtered through the use of qualifiers that can be themselves XPath expressions. For instance, the XPath expression:

```
/descendant::a[following-sibling::b]/child::c
```

A detailed taxonomy of schema languages for XML can be found in [Murata et al. 2005].

$$
\begin{aligned}
\mathit{path} &::= /\mathit{path} \mid \mathit{path}/\mathit{path} \mid \mathit{path}[\mathit{qualifier}] \mid \mathit{path} \cup \mathit{path} \mid \mathit{path} \cap \mathit{path} \mid \mathit{path} \text{ except } \mathit{path} \mid \mathit{step} \\
\mathit{qualifier} &::= \mathit{path} \mid \text{not } \mathit{qualifier} \mid \mathit{qualifier} \text{ and } \mathit{qualifier} \\
&\quad \mid \mathit{qualifier} \text{ or } \mathit{qualifier} \\
\mathit{step} &::= \mathit{axis}::\mathit{test} \\
\mathit{axis} &::= \text{self} \mid \text{child} \mid \text{parent} \mid \text{descendant} \mid \text{ancestor} \mid \text{following-sibling} \mid \text{preceding-sibling} \\
&\quad \mid \text{following} \mid \text{preceding} \mid \text{ancestor-or-self} \mid \text{descendant-or-self} \\
\mathit{test} &::= \mathit{tag} \mid *
\end{aligned}
$$

Fig. 2. Syntax of Considered XPath Expressions.

navigates from the root of the document to all descendant nodes labeled "a", and retains only those which have at least one following sibling node labeled "b", then it finally selects and returns the set of all children nodes labeled "c" of those "a" nodes. The above expression is absolute since it begins with a leading "/" that specifies that the initial context node for the evaluation of the expression is the root node of the document tree. A relative XPath expression (without the leading "/") can be evaluated from any given context node in the tree.

3.1. Syntax and Semantics of XPath Expressions

In this paper, we consider the XPath fragment whose syntax is shown in Figure 2, and that contains all XPath features for navigating forward, backward and recursively through nodes of the document. The fragment also captures the fact that nodes can be filtered using qualifiers, which are boolean expressions between brackets that can test the existence or absence of nodes.

We recall the set-theoretic semantics of XPath expressions from [Wadler 2000]. Given a context node x in an XML document tree, the evaluation of an XPath expression returns a set of nodes. The formal semantics function $\mathcal{S}_p$ defines the set of nodes returned by an XPath expression:

$$
\begin{aligned}
\mathcal{P}[\![\cdot]\!]_{\cdot} &: \mathit{path} \times \mathit{Node} \longrightarrow \text{Set}(\mathit{Node}) \\
\mathcal{P}[\![/\mathit{path}]\!]_x &= \mathcal{P}[\![\mathit{path}]\!]_{\mathit{root}()} \\
\mathcal{P}[\![\mathit{path}/\mathit{path}']\!]_x &= \{x_2 \mid x_1 \in \mathcal{P}[\![\mathit{path}]\!]_x \wedge x_2 \in \mathcal{P}[\![\mathit{path}']\!]_{x_1}\} \\
\mathcal{P}[\![\mathit{path}[\mathit{qualifier}]]\!]_x &= \{x_1 \mid x_1 \in \mathcal{P}[\![\mathit{path}]\!]_x \wedge \mathcal{Q}[\![\mathit{qualifier}]\!]_{x_1}\} \\
\mathcal{P}[\![\mathit{path} \cup \mathit{path}']\!]_x &= \mathcal{P}[\![\mathit{path}]\!]_x \cup \mathcal{P}[\![\mathit{path}']\!]_x \\
\mathcal{P}[\![\mathit{path} \cap \mathit{path}']\!]_x &= \mathcal{P}[\![\mathit{path}]\!]_x \cap \mathcal{P}[\![\mathit{path}']\!]_x \\
\mathcal{P}[\![\mathit{path} \text{ except } \mathit{path}']\!]_x &= \mathcal{P}[\![\mathit{path}]\!]_x \setminus \mathcal{P}[\![\mathit{path}']\!]_x \\
\mathcal{P}[\![\mathit{axis}::\mathit{tag}]\!]_x &= \{x_1 \mid x_1 \in \mathcal{A}[\![\mathit{axis}]\!]_x \wedge \mathit{name}(x_1) = \mathit{tag}\} \\
\mathcal{P}[\![\mathit{axis}::*]\!]_x &= \{x_1 \mid x_1 \in \mathcal{A}[\![\mathit{axis}]\!]_x\}
\end{aligned}
$$

The function $\mathcal{S}_q$ defines the semantics of qualifiers that basically state the existence (or absence) of one or more nodes from a context node x:

An interpretation of XPath expressions in terms of modal logic can be found in [Genevès et al. 2007].

$$
\begin{aligned}
\mathcal{Q}[\![\cdot]\!]. : \mathit{Qualifier} \times \mathit{Node} &\longrightarrow \mathit{Boolean} \\
\mathcal{Q}[\![\mathit{path}]\!]_x &= \mathcal{P}[\![\mathit{path}]\!]_x \neq \emptyset \\
\mathcal{Q}[\![\text{not } \mathit{qualifier}]\!]_x &= \neg \mathcal{Q}[\![\mathit{qualifier}]\!]_x \\
\mathcal{Q}[\![\mathit{qualifier} \text{ and } \mathit{qualifier}']\!]_x &= \mathcal{Q}[\![\mathit{qualifier}]\!]_x \wedge \mathcal{Q}[\![\mathit{qualifier}']\!]_x \\
\mathcal{Q}[\![\mathit{qualifier} \text{ or } \mathit{qualifier}']\!]_x &= \mathcal{Q}[\![\mathit{qualifier}]\!]_x \vee \mathcal{Q}[\![\mathit{qualifier}']\!]_x
\end{aligned}
$$

The function $\mathcal{S}_a$ gives the denotational semantics of XPath axes that navigate in trees:

$$
\begin{array}{lcl}
\mathcal{A}[\![\cdot]\!]. & : & \mathit{Axis} \times \mathit{Node} \longrightarrow \text{Set}(\mathit{Node}) \\
\mathcal{A}[\![\text{child}]\!]_x & = & \mathit{children}(x) \\
\mathcal{A}[\![\text{parent}]\!]_x & = & \mathit{parent}(x) \\
\mathcal{A}[\![\text{descendant}]\!]_x & = & \mathit{children}^+(x) \\
\mathcal{A}[\![\text{ancestor}]\!]_x & = & \mathit{parent}^+(x) \\
\mathcal{A}[\![\text{self}]\!]_x & = & \{x\} \\
\mathcal{A}[\![\text{descendant-or-self}]\!]_x & = & \mathcal{A}[\![\text{descendant}]\!]_x \cup \mathcal{A}[\![\text{self}]\!]_x \\
\mathcal{A}[\![\text{ancestor-or-self}]\!]_x & = & \mathcal{A}[\![\text{ancestor}]\!]_x \cup \mathcal{A}[\![\text{self}]\!]_x \\
\mathcal{A}[\![\text{preceding}]\!]_x & = & \{y \mid y \ll x\} \setminus \mathcal{A}[\![\text{ancestor}]\!]_x \\
\mathcal{A}[\![\text{following}]\!]_x & = & \{y \mid x \ll y\} \setminus \mathcal{A}[\![\text{descendant}]\!]_x \\
\mathcal{A}[\![\text{following-sibling}]\!]_x & = & \{y \mid y \in \mathit{child}(\mathit{parent}(x)) \wedge x \ll y\} \\
\mathcal{A}[\![\text{preceding-sibling}]\!]_x & = & \{y \mid y \in \mathit{child}(\mathit{parent}(x)) \wedge y \ll x\}
\end{array}
$$

in which $\mathit{root}()$, $\mathit{children}(x)$ and $\mathit{parent}(x)$ are primitives for navigating unranked trees, $\ll$ is the ordering relation ($x \ll y$ holds if and only if the node x is before the node y in the depth-first traversal order of the tree), and $\mathit{name}()$ returns the label of a given tree node.

3.2. Applications of XPath

Since search, selection and extraction of information are essential for any XML processing task, XPath happens to be a core component of XML technologies. In particular, XPath plays a major role in the main XML technologies: XSLT, XML Schema, XForms, and XQuery.

XSLT. XSLT is the W3C standard language for transforming XML documents into other XML documents. XSLT relies on XPath for selecting parts of the input documents to be transformed. Specifically XPath is used for two different purposes in XSLT: (1) identifying source nodes to which transformation rules apply, by the means of patterns to be matched with nodes of the input document; (2) extracting information, in particular for selecting the next input nodes to be transformed, computing boolean conditions, and producing output text in the result tree.

XML Schema. XML Schema is a W3C standard for the description of document types. An XML schema can be used to express a set of rules to which an XML document must conform in order to be considered *valid* according to that schema. XML Schema provides several XPath-based features for describing uniqueness constraints and corresponding references constraints.

XForms. XForms is a W3C standard for the specification of a data processing model for XML data and user interfaces for the XML data, such as web forms. XForms notably uses XPath expressions as queries for addressing and identifying fields within the form and the user-submitted data. In particular, it uses XPath expressions to bind input controls to particular parts of the form's data model.

XQuery. XQuery is the W3C standard language for querying collections of XML data. XQuery is heavily based on XPath expressions to address specific parts of an XML document. It supplements this with a SQL-like "`FLWOR` expression" for performing joins. A `FLWOR` expression is constructed from the five clauses after which it is named: `FOR`, `LET`, `WHERE`, `ORDER BY`, `RETURN`.

Java/Javascript. In addition, Java and Javascript programs now make extensive use of XPath expressions through the Document Object Model (DOM).

4. PATH REASONING

4.1. Path Inconsistencies

The XPath language provides a succinct yet very expressive notation for expressing relations between nodes in trees. The possibility of expressing complex relations like first-order definable relations [Marx 2004] is counterbalanced by the fact that inconsistencies can easily be introduced. Typical XPath errors are often introduced due to the expressive power of XPath for expressing forward, backward and recursive navigation in trees. A given XPath expression e is said to be *inconsistent* iff, for any document (tree), the evaluation of e returns an empty set of nodes (e.g. using the formal semantics presented in Section 3.1, $\forall t, \forall x \in t, \mathcal{P}[\![e]\!]_x = \emptyset$, where x is some node in the tree t). Inconsistent XPath expressions usually contain a contradiction between two navigation steps (either successive or not). XPath expressions contained in qualifiers may also contradict expressions used for selection.

It may happen that navigational information contained in a given XPath expression contradicts some constraints described by a schema. Therefore, even a consistent XPath expression may systematically yield an empty result whenever the set of documents over which it is supposed to be evaluated is constrained by a schema.

A schema defines a restricted set of documents that conform to some constraints, usually expressed by the means of regular expressions. These constraints restrict the admissible elements and the way they can be composed. Widespread notations for schemas include DTD, XML Schema, and Relax NG. We say that an XPath expression e is *inconsistent in the presence of a schema* S iff the evaluation of e returns the empty set for all documents valid with respect to S. This kind of inconsistencies is frequent since XPath expressions and schemas are entities that are updated independently.

In that case, the XPath expression will always return an empty sequence of nodes no matter what the actual document instance valid for S is.

Detecting inconsistent XPath expressions (in the presence or absence of a schema) is crucial for any static analysis of a host language for XPath such as XSLT, XQuery, XForms or XML Schema. Since XPath plays a central role in all these languages, offering the capability of manipulating and analyzing XPath expressions in programming environments, notably IDEs, simplifies the development and verification of a large class of applications. For instance, inconsistent XSLT patterns or XQuery subpaths mean that dead code can be eliminated: if an XPath expression is statically detected as inconsistent, one can avoid evaluating it at runtime in order to save resources. In addition, all code that is supposed to treat the result of an inconsistent XPath expression can be safely eliminated. The detection of dead code in XQuery programs presented in Section 6 uses the detection of inconsistent XPath expressions as an essential component.

4.2. Logical Resolution: Background Theory

The problem of determining whether an XPath expression is inconsistent in the presence of a schema, for the XPath language and the schemas considered in this article, is known to be exponential-time complete from [Benedikt et al. 2008] and more precisely

decidable in time $2^{\mathcal{O}(n)}$ from [Genevès et al. 2007]. The decision procedure determines the existence (or absence) of a tree (a document) that satisfies both the constraints expressed by the schema and the structural requirements assumed by the XPath expression. Specifically, the decision procedure assumes a logical representation of the problem in terms of a modal logic (specifically a μ-calculus) of trees. This representation is a logical formula that combines all the requirements of the problem. The decision procedure then consists in a logical satisfiability-testing algorithm that looks for a finite tree that satisfies the logical formula. If a finite tree is found, then the formula is satisfiable, otherwise it is unsatisfiable. The decision procedure works as a fixpoint computation: it explores the universe of trees in a bottom-up fashion, repeatedly considering new tree nodes, until all the relevant tree nodes have been explored. The decision procedure is sound and complete, which means that it is exact (in particular the search is exhaustive). The difficult aspect from an algorithmic point of view is that the search universe in which a tree is looked for is very large. To help overcoming this difficulty, semi-implicit representation techniques are used. These techniques avoid the explicit enumeration of tree nodes. They abstract over sets of tree nodes using boolean expressions, making it possible to manipulate very large sets of nodes through operations over small boolean expressions. See [Genevès et al. 2008] for more details concerning the main solver implementation techniques as well as the specific optimizations developed.

4.3. Supported Features via Approximations

Certain XPath features such as data value equality ("joins") and comparisons between expressions that count node occurrences are known to cause undecidability of the inconsistency check for XPath expressions [Benedikt et al. 2008]). While an exact analysis of expressions using these features is out of reach, their usage does not prevent an approximate (sound but incomplete) analysis. For example, an XPath expression of the form:

$$path[path_1 = path_2] \tag{1}$$

is analyzed via the approximation:

$$path[path_1 \text{ and } path_2] \tag{2}$$

More precisely, from the XPath specification [Clark and DeRose 1999], the expression (1) requires that a least one node selected by $path_1$ has a string-value that matches the string-value of a node selected by $path_2$.

The approximation (2) that we consider for (1) requires that the constraints described by $path_1$ as well as those described by $path_2$ exist, or in other terms, both $path_1$ and $path_2$ evaluate to non-empty sets of nodes. The approximation is necessary but not sufficient for the equality in (2) to hold. It does not check the actual data values but translates the necessary structural requirements for the equality to hold.

The other approximation that we consider concerns counting expressions. XPath expressions of the form:

$$path[count(path_1) \triangle count(path_2)] \tag{3}$$

where the function "$count(path)$" returns the number of occurrences of nodes selected by the XPath expression "$path$" [Clark and DeRose 1999] and $\triangle \in \{<, >, =, \leq, \geq\}$, are also abstracted over by (2).

The string-value of a node is the concatenation of all descendant text nodes, in document order. See [Clark and DeRose 1999] for more details.

4.4. Logical Resolution: Practice

In practice, for performing the inconsistency analysis, the system takes a given XPath expression and a schema, constructs the corresponding approximations when necessary, translates the result into a logical representation owing to the compilers described in [Genevès et al. 2007], and then solves the formula for satisfiability with the logical solver of [Genevès et al. 2007], as described above.

For typical cases involving only an XPath expression or an XPath expression and a small schema, the decision procedure performs in several milliseconds. For hard cases involving a highly complex schema like the XHTML DTD, the total number of different tree nodes to be considered is more than the square of the number of atoms in the universe. Even for these cases, it was shown in [Genevès et al. 2007; Genevès et al. 2009] that the decision procedure performs in less than 3 seconds. This surprising efficiency is partly due to the use of symbolic techniques as well as other advanced implementations techniques and optimizations [Genevès et al. 2007]. The advantage of a high efficiency is that it permits equipping IDEs with automated detection of inconsistent XPath expressions when the user clicks a button, as presented in the next section.

4.5. Path Redundancies

The aforementioned procedure for determining inconsistency of XPath expressions can also be used to check for a related property: XPath inclusion. The XPath inclusion problem consists in determining, whether, for any document tree, the set of nodes returned by the evaluation of an XPath expression path_1 is included in the one returned by the evaluation of another XPath expression path_2, starting from any (common) context node, e.g.:

$$\forall t, \forall x \in t, \mathcal{P}[\![\mathit{path}_1]\!]_x \subseteq \mathcal{P}[\![\mathit{path}_2]\!]_x \tag{4}$$

Technically, when negation is part of the query language, checking XPath expression inclusion reduces to checking XPath expression inconsistency:

$$\forall t, \forall x \in t, \mathcal{P}[\![\mathit{path}_1 \cap \text{not } (\mathit{path}_2)]\!]_x = \emptyset \tag{5}$$

We use the aforementioned procedure to check for XPath expression inclusion in order to eliminate redundancies from XPath expressions, and automatically rewrite them into simpler (more succinct) ones. For example, the expression

$$/a[child :: *]/c \tag{6}$$

is automatically simplified as

$$/a/c \tag{7}$$

by the rule QUALIF3 shown on Figure 3. The simplification rules are expressed as refactoring rules, that rely on inclusion tests in order to check for the presence of redundancies. Figure 3 presents the simplification rules. Each rule of the form:

$$\frac{H}{e \rightsquigarrow e'}$$

can be read as follows: provided that hypothesis H is satisfied, expression e is automatically rewritten into expression $e2$. The hypothesis H in the premises of the rule typically uses an inclusion or equivalence test. An inclusion test between two expressions (whether XPath expressions or qualifiers), denoted $e_1 \leq e_2$, is simply translated in terms of a logical implication which is sent to the solver [Genevès et al. 2007]. An equivalence test, denoted by $e_1 \equiv e_2$, simply amounts to a double inclusion test $e_1 \leq e_2$ and $e_2 \leq e_1$.

Correctness of these rules is proved by structural induction using the set-theoretics semantics of XPath expressions given in Section 3.1.

The process of simplification is applied repeatedly to an input XPath expression until no more simplification can be found. These rules are general: they are used in the presence or absence of schemas.

This simplification/refactoring process is very useful for gaining efficiency at runtime because most of the XPath evaluation engines apply a step by step evaluation (inspired from the semantics of XPath expressions given in Section 3.1), and they are not aware of the potential irrelevance of evaluating some subexpressions. Performing this automated simplification once statically removes the costs of unnecessary evaluations at (every) runtime. These optimizations have been proved useful with engines such as Xalan [Xalan 2013] and Libxml [Veillard 2013] (see [Genevès and Vion-Dury 2004] for practical experiments on this topic).

$$\textsc{union}\quad \frac{path' \leq path}{path \cup path' \longrightarrow path} \qquad \textsc{intersect}\quad \frac{path' \leq path}{path \cap path' \longrightarrow path'}$$

$$\textsc{qualif1}\quad \frac{qualifier_1 \leq qualifier_2}{path[qualifier_1 \text{ and } qualifier_2] \longrightarrow path[qualifier_1]}$$

$$\textsc{qualif2}\quad \frac{qualifier_1 \leq qualifier_2}{path[qualifier_1 \text{ or } qualifier_2] \longrightarrow path[qualifier_2]}$$

$$\textsc{qualif3}\quad \frac{path_2 \leq qualifier}{path_1[qualifier]/path_2 \longrightarrow path_1/path_2} \qquad \textsc{qualif4}\quad \frac{path_2 \leq qualifier}{path_1[qualifier]/path_2 \longrightarrow path_1/path_2}$$

$$\textsc{prefix}\quad \frac{path_1 \equiv path_2 \quad \triangle \in \{\cup, \cap\}}{(path_1/path_3) \triangle (path_2/path_4) \longrightarrow path_2/(path_3 \triangle path_4)}$$

$$\textsc{prefix'}\quad \frac{path_1 \equiv path_2 \quad \triangle \in \{\cup, \cap\}}{(path_1[qualifier_1]) \triangle (path_2/[qualifier_2]) \longrightarrow path_1[qualifier_1 \triangle qualifier_2]}$$

$$\textsc{suffix}\quad \frac{path_3 \equiv path_4 \quad \triangle \in \{\cup, \cap\}}{(path_1/path_3) \triangle (path_2/path_4) \longrightarrow (path_1 \triangle path_2)/path_4}$$

$$\textsc{suffix'}\quad \frac{qualifier_1 \equiv qualifier_2 \quad \triangle \in \{\cup, \cap\}}{(path_1[qualifier_1]) \triangle (path_2[qualifier_2]) \longrightarrow (path_1 \triangle path_2)[qualifier_1]}$$

Fig. 3. Rules for Automated Simplification.

5. AUGMENTED IDE

As a proof of concept, we have integrated this automated analysis inside an IDE: we have equipped the XQuery Development Toolkit (XQDT) [XQDT 2011] with capabil-

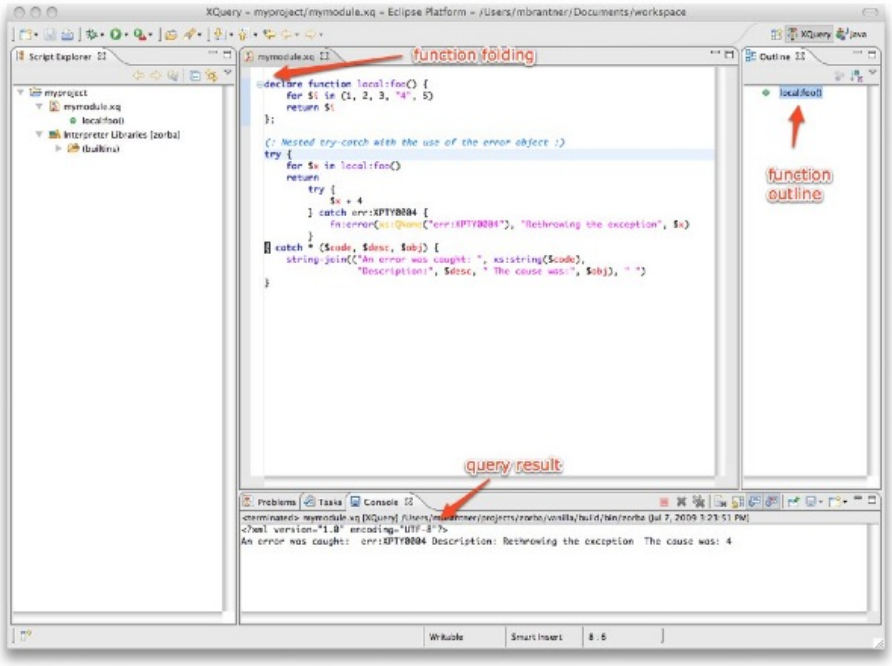

Fig. 4. Overview of XQDT user interface.

ities of statically detecting inconsistent XPath expressions. XQDT is a plugin for the Eclipse environment that provides support for XQuery 1.1. In particular XQDT provides code completion and code templates, as-you-type validation, and integration with existing XQuery evaluation engines. A screenshot of XQDT is given in Figure 4.

5.1. Integration Principle

We have developed a plugin extension that takes an XPath expression e and a schema S as parameters and checks for the inconsistency of e in the presence of S. The analysis functions mark inconsistent XPath expressions with syntax coloring capabilities offered by the IDE plugin. For this to be possible, the IDE plugin interacts with the plugin extension in the following manner:

(1) the abstract syntax tree of the program is first analyzed in order to identify XPath expressions;
(2) the evaluation context of each XPath expression is built: because some XPath host languages (like XSLT or XQuery) allow variables to be defined and then used in XPath expressions, this step is necessary for correctly replacing variables occurring in XPath expressions by their definition;
(3) when the static verification is triggered, each XPath expression and its evaluation context as well as the schema chosen by the programmer are transmitted to the plugin extension;
(4) once the analysis is performed the plugin extension returns information (line number, character index) in order to mark inconsistent XPath expressions in the user interface.

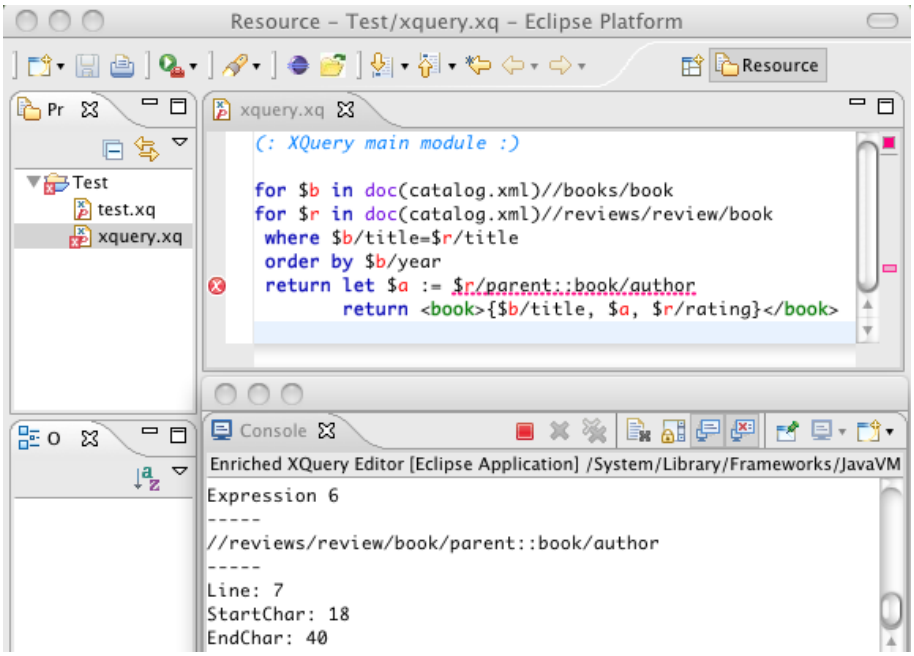

Fig. 5. Static Analysis of XPath Expressions in Action.

5.2. Enriched Programming Experience

From the programmer's point of view, the augmented XQDT plugin can be used just as the usual XQDT plugin. The only difference happens when a given XQuery program is opened through the user interface. Two new buttons are then offered to the programmer. The first one allows him to choose a given schema (notice that this is optional: by default no schema is assumed). The second button allows the programmer to trigger the static analysis of XPath expressions which marks inconsistent XPath expressions, in the same manner as badly typed Java statements are marked in the classic Eclipse environment for editing Java programs.

The user interface of the plugin extension is shown in Figure 5. The screenshot shows an XQuery example where an XPath expression is automatically identified and marked as inconsistent (independently of any schema). In this case, the XPath expression "`$r/parent::book/author`" is trivially judged inconsistent by the analysis since if we replace "`$r`" by its definition we obtain an expression of the form:

```
//reviews/review/book/parent::book/author
```

that at some point attempts to navigate from a node labeled "`review`" to children nodes labeled "`book`" and then going back to the parent node labeled "`book`", which contradicts the previous steps according to which this parent node is labeled "`review`". For this reason, evaluating this XPath expression always yields an empty set of nodes, even independently from any schema. General XPath expression inconsistencies are not so trivial to detect, especially those involving schema information. This is why inconsistent XPath expressions are clearly marked à la Eclipse in the user interface: they are underlined in red and marked with red icons both in the left gutter and next to the scroll bar on the right in order to inform the programmer.

6. DEAD-CODE ANALYSIS FOR XQUERY

In this section, we present a static analysis of XQuery programs, based on the detection of inconsistent XPath expressions (with and without schema information) in order to automatically detect and eliminate dead code. We first introduced XQuery programs, and in particular the fragment that we consider, and then the analysis of dead code.

6.1. XQuery Programs

An XQuery program basically takes one (or possibly several) XML document as input, performs some computation based on its tree view, and finally outputs a result in the form of another XML document. The core of the XQuery language is composed of XPath expressions that make it possible to navigate in the document tree and extract nodes that satisfy some conditions. For instance, a simplistic XQuery program is:

```
<ul>
 {
  for $x in /descendant::book return
    if $x/year>2008 then <li>$x/title<li> else ()
 }
</ul>
```

where the `for` loop uses the XPath expression `/descendant::book` that traverses the whole input XML document looking for `book` elements. The `for` loop iterates over all these elements, and for each of them, returns the value of the `title` subelement, provided the year is greater than 2008. Executing this program produces an XML tree as output, whose root element is named "`ul`", and whose content is populated by the execution of the loop, that creates an XHTML-like list of book titles published after 2008.

In the remaining, we consider a fragment of the XQuery programming language, whose syntax is given in Figure 6 (the semantics is described in [Boag et al. 2007]). This fragment focuses on the core aspects of XQuery and in particular XPath expressions for navigating and extracting information from XML trees. This fragment reuses XPath expressions presented in Figure 2.

6.2. XQuery Dead Code

In order to illustrate the practical relevance of our approach, consider the following XQuery program:

```
<para>
  {
    for $x in //body//switch
    where $x/animateMotion
    return $x/*
  }
</para>
```

It is intended to be evaluated over SMIL documents. Specifically, it has been written against the schema defining SMIL 1.0 documents. When applied to such a document, it returns all children of `switch` elements that have at least one `animateMotion` child, wrapped in a `para` element.

SMIL is the standard language for expressing synchronized multimedia documents as found in e.g., MMS mobile phone messages, and more generally on the web [Hoschka 1998].

$$
\begin{array}{rcl}
e & ::= & \\
 & & \texttt{()} \\
 & | & e\texttt{,}e \\
 & | & \texttt{<}tag\texttt{>}e\texttt{</}tag\texttt{>} \\
 & | & \texttt{element}\{e\}\{e\} \\
 & | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & | & \texttt{for}\ \$var\ \texttt{in}\ e\ (\texttt{where}\ e)?\ \texttt{return}\ e \\
 & | & \texttt{let}\ \$var\ \texttt{:=}\ e\ (\texttt{where}\ e)?\ \texttt{return}\ e \\
 & | & (\texttt{some} \mid \texttt{every})\ \$var\ \texttt{in}\ e\ \texttt{satisfies}\ e \\
 & | & \texttt{typeswitch}\ \texttt{(}e\texttt{)}\ cases \\
 & | & \$var \\
 & | & path \\
 & | & \$var/path \\
 & | & /path \\
 & | & e\ op\ e \\
 & | & f(e,...,e) \\
 & | & arithmeticExpr \\
cases & ::= & \\
 & & \texttt{default return}\ e \\
 & | & \texttt{case}\ Type\ \texttt{return}\ e\ cases
\end{array}
$$

Fig. 6. Syntax of XQuery Programs.

This code portion may be reused in the context of SMIL 2.0 documents. However, in contrast to SMIL 1.0, the occurrence of `animateMotion` is not permitted as a chid of `switch` in SMIL 2.0. In this case, the XPath expression in the `where` clause is inconsistent, and therefore the whole `for` loop is dead code. We explain how we make this static analysis automatic for a given XQuery program and a given schema in the next subsections.

6.3. Dead-Code Analysis based on Path-Error Detection

We consider a given XQuery program P and a schema S that describes constraints over the set of documents that can serve as input to P. For each XPath expression occurring in P, we check whether it is inconsistent in the presence of S. In that case, we know statically that there is no need to evaluate the XPath expression at runtime. Furthermore, we also know that all XQuery instructions that depend on this XPath expression (dead code) may be removed.

This analysis is sound and complete over the XPath navigational fragment shown in Figure 2. In order for the analysis to scale to programs with more complex features, we make several conservative approximations. First, we abstract over XPath features that make satisfiability undecidable (such as data value comparisons), as described in Section 4.3. Second, we consider that XPath expressions still return sets of nodes (as in XPath 1.0) instead of node sequences (as in XPath 2.0 and XQuery). These approximations preserve soundness of our approach: if dead code is detected, it can be safely eliminated as this is really dead code.

6.4. Static Code Refactoring

Each XPath expression which is found inconsistent indicates dead code. We perform a code dependency analysis that propagates this information in order to detect and eliminate dead code from an XQuery program. The analysis consists of inference rules

$$\frac{S,\Gamma : e_1 \longrightarrow ()}{S,\Gamma : \texttt{element}\{e_1\}\{e_2\} \longrightarrow ()} \qquad \frac{S,\Gamma : e_i \longrightarrow e_i' \qquad i = 1,2 \qquad e_1' \neq ()}{S,\Gamma : \texttt{element}\{e_1\}\{e_2\} \longrightarrow \texttt{element}\{e_1'\}\{e_2'\}}$$

$$\frac{S,\Gamma : e_i \longrightarrow () \qquad i = 1,3}{S,\Gamma : \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \longrightarrow ()} \qquad \frac{S,\Gamma : e_1 \longrightarrow () \qquad S,\Gamma : e_3 \longrightarrow e_3' \qquad e_3' \neq ()}{S,\Gamma : \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \longrightarrow e_3'}$$

$$\frac{S,\Gamma : e_i \longrightarrow e_i' \qquad i = 1,2,3 \qquad e_1' \neq ()}{S,\Gamma : \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \longrightarrow \texttt{if } e_1' \texttt{ then } e_2' \texttt{ else } e_3'}$$

$$\frac{S,\Gamma : e_1 \longrightarrow ()}{S,\Gamma : \texttt{for } \$var \texttt{ in } e_1 \ (\texttt{where } e_2)? \texttt{ return } e_3 \longrightarrow ()}$$

$$\frac{S,\Gamma \cup (\$var, e_1) : e_3 \longrightarrow ()}{S,\Gamma : \texttt{for } \$var \texttt{ in } e_1 \ (\texttt{where } e_2)? \texttt{ return } e_3 \longrightarrow ()}$$

$$\frac{S,\Gamma : e_1 \longrightarrow e_1' \qquad e_1' \neq () \qquad S,\Gamma \cup (\$var, e_1') : e_2 \longrightarrow ()}{S,\Gamma : \texttt{for } \$var \texttt{ in } e_1 \texttt{ where } e_2 \texttt{ return } e_3 \longrightarrow ()}$$

Fig. 7. XQuery Refactoring Rules.

of the form:

$$\frac{H}{S,\Gamma : e \longrightarrow e'}$$

Such a rule means that the original program e is rewritten into another program e' assuming some hypothesis H in the context of a schema S and a variable environment Γ. The benefit of this rewriting is that e' is dead-code free. The rewriting is also safe in the sense that it preserves the semantics of the original program: executing the rewritten program yields the same result than executing the original program. The only difference is that e' is smaller than e in terms of code size, and thus potentially executes faster.

One of the most basic rules consists in replacing an inconsistent XPath expression by the empty node sequence, as follows:

$$\frac{\neg\text{satisfiable}(path, S)}{S,\Gamma : path \longrightarrow ()}$$

where the predicate satisfiable$(path, S)$ in the hypothesis is the boolean test directly performed by the logical solver [Genevès et al. 2007]. This rewriting is extended to other XQuery statements. Figure 7 details the rewriting principle for three main XQuery constructs, namely the instruction for generating elements, the "if" statement and the "for" loop. For instance, the first rule eliminates the instruction `element`$\{e_1\}\{e_2\}$ (rewrites it to the empty sequence) provided the expression e_1 rewrites itself to the empty sequence. The third rule eliminates a whole `if` statement whenever both the `if` condition and the `else` clause rewrite to the empty sequence. Rules for other XQuery constructs follow the same principle and are similar.

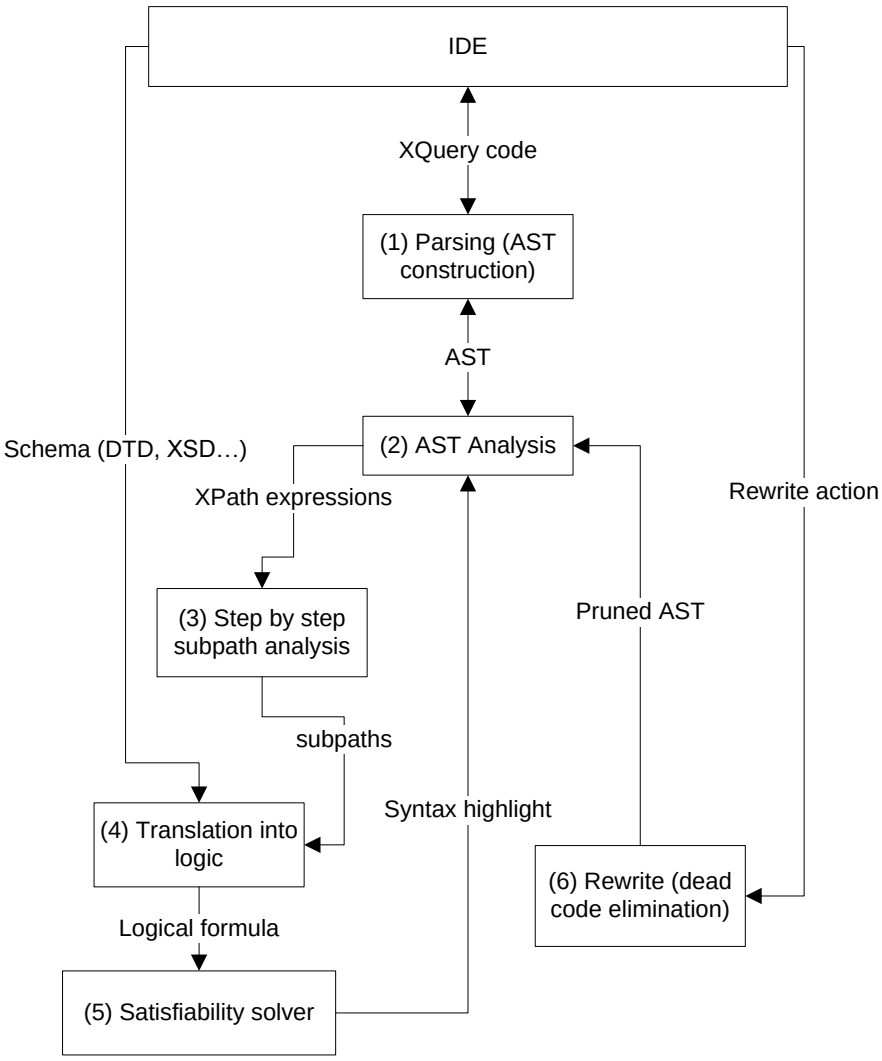

Fig. 8. Code Analysis Diagram.

6.5. Syntax Highlighting & Code Refactoring

The typical integrated development environment allows one to open an XQuery program and to associate with it a schema. The code analysis process is illustrated in Figure 8. First, the program is parsed to build an abstract syntax tree (step 1 in Figure 8). The abstract syntax tree (AST) analysis phase consists in extracting all the XPath expressions from the program and checking their satisfiability individually (steps 2 to 5). Then, in a second step, these XPath expressions are combined with the schema, and checked again for satisfiability (steps 2 to 5 again). This if for clearly distinguishing inconsistent XPath expressions (e.g. `child::a/child::b[parent::c]`) from inconsistent

A variety of schemas are actually supported including DTDs, XML Schemas and Relax NG definitions (see [Genevès et al. 2007] for details).

XPath expressions in the presence of the schema. Each kind of inconsistent XPath expression is marked differently in the AST. This makes it possible to inform the programmer, by underlining the empty XPath expressions in a different color depending on the origin of the inconsistency (self-contradiction or inconsistency in the presence of the given schema). More specifically, each XPath expression is considered as a sequence of basic navigation steps possibly with qualifiers. The first step is analyzed. Then each additional step is successively appended to this initial step and the resulting XPath expression is analyzed in turn (step 3). This makes it possible to identify precisely where the error has been introduced in the XPath expression. For instance, in the previous example, this step by step subpath analysis identifies the qualifier `parent::c` as causing the error. Likewise, inconsistencies between different XPath expressions can be detected (e.g. WHERE clauses are supported as intersections of XPath expressions already supported by the fragment described in Section 2).

Whenever an inconsistent XPath expression is found, a refactoring command is provided to the IDE user. When this command is triggered, the AST is pruned using the rules presented earlier (step 6), and the new XQuery program is provided to the user.

7. RELATED WORKS

To the best of our knowledge, our work is the first to provide an IDE equipped with XPath expression reasoning techniques such as a precise static detection of inconsistent XPath expressions. As a consequence, in this context, other IDEs (even supporting syntax verification and/or runtime debugging features) do not match the static analysis precision and capabilities of the work presented here.

A line of research which relies on static analysis techniques similar to ours is followed in [Marian and Siméon 2003; Benzaken et al. 2006]. The underlying idea of these works is quite simple: given a query q over a document d, the subtrees of d not necessary to evaluate q are pruned, thus obtaining a smaller document d'. Then q is executed over d', hence avoiding to allocate and process nodes that will never be reached by navigational specifications in q. For this purpose, a smaller schema containing only the relevant parts of the document with respect to the query is inferred from the initial schema. The idea we pursue here differs in that we seek to prune the XQuery program itself (not the schema). The two optimizations are in fact complementary, and a perspective consists in combining them.

The system we propose involves solving the satisfiability problem for an expressive fragment of XPath expressions, a task which is known to be very complex from a computational point of view [Benedikt et al. 2008]. Early techniques for the containment of tree queries were extensively studied using tree patterns in the literature [Amer-Yahia et al. 2001; Miklau and Suciu 2004]. The corresponding containment and minimization methods focused on much smaller XPath fragments. Typically the work found in [Amer-Yahia et al. 2001] focuses on the XPath fragment using only the operators "descendant" and positive qualifiers (named $\mathrm{XP}^{\{[\,],//\}}$) for which a polynomial-time containment algorithm is provided. The work found in [Miklau and Suciu 2004] focuses on the fragment using only the operators "*", "child", "descendant" and positive qualifiers (named $\mathrm{XP}^{\{*,//,[\,]\}}$). In particular, containment for the latter fragment is shown to be coNP-complete in [Miklau and Suciu 2004], where the containment mapping technique relies on a polynomial time tree homomorphism algorithm, which gives a sufficient but not necessary condition for containment of $\mathrm{XP}^{\{*,//,[\,]\}}$ in general. In comparison, the method described in [Genevès et al. 2007] is sound and complete and applies to a much larger XPath fragment supporting for instance reverse XPath axes and negation inside qualifiers. Therefore, for solving this problem, we build on our previous work on the static analysis of XPath and reuse the satisfiability solver

developed in [Genevès et al. 2007; Genevès and Layaïda 2010], that was tested on real life use cases (such as XHTML and MathML types) and complex queries (involving recursive and backward navigation) [Genevès et al. 2009]. This logic-based approach proved extensible as it served as a ground for further extending the set of supported query language features, notably with counting and interleaving operators [Barcenas et al. 2011], attributes [Genevès et al. 2012], functions [Gesbert et al. 2011], and operators for graphs [Chekol et al. 2012a; Chekol et al. 2012b].

A closely related area of work concerns the evolution of schemas and their impact on queries. XML schema collections do constantly evolve, therefore some previously written queries in languages such as XPath and XQuery may become inconsistent with the latest versions of the schemata. The techniques we propose to detect XPath expression inconsistencies, to automatically simplify queries, and automatically remove dead code can be used in this context. Substantial investigations have already been carried out in this specific context.

In particular, the work found in [Genevès et al. 2011] is concerned with the representation of schema changes in terms of logic in order to precisely infer the behavior of XPath queries over successive schema versions. However this work is limited to the identification of the impact of schema changes on XPath queries, but does not help in reformulating them.

Researchers provide solutions for the problem of schema evolution in the presence of relational queries [Curino et al. 2008; Curino et al. 2013]. The main difference with the present work is that the relational framework is a much more constrained than the XML context, which introduces structure and order, and allows for more flexible constraints. Furthermore, [Curino et al. 2008] relies on a notion of mapping which does not exist in the context of software engineering for XML.

[Termehchy et al. 2012] introduces a notion of design independence in the context of schema evolution. The focus of the paper is on the construction of queries that remain robust to schema changes. However the considered query language (with queries in the form of a bag of terms) is incomparable with XPath and XQuery considered in the present article. [Truong et al. 2012] explores this notion of structure independence by introducing a non-directional XPath axis called the neighborhood axis. However, this article focuses on performance of evaluation and not on static analysis.

8. CONCLUSION

We have presented new results in the static analysis and code refactoring for core XML technologies: the static detection of XQuery dead code with the automatic elimination/refactoring of the corresponding code; and the elimination of redundancies for the automatic simplification of XPath expressions. We have combined these developments in order to provide the first IDE equipped with XPath reasoning capabilities. As a proof of concept, we developed a plugin extension of XQDT that considers XPath expressions as first-class constructs and is capable of underlining inconsistent XPath expressions in the same manner as badly typed Java statements in the classic Eclipse environment for editing Java programs.

The tool integrates a formal prover of properties on schemas and XPath expressions for assisting programmers in the writing and updating of XQuery code against complex XML schemas. This analysis is plugged on the syntactic analyzer of the IDE. It intercepts the modifications of the abstract syntax tree maintained by the editor following the editing operations performed by the user. It then identifies and solves reasoning tasks about XPath expressions and schemas on the logical solver side. The results of these analyses are then retranslated in terms of new decorations of the abstract syntax tree for notifying precisely errors and dead code interactively to the user via syntax coloring.

We also describe the architecture and mechanisms used for the integration of the logical solver with the parser and user interface features of an IDE. This general scheme can be generalized to operate with any other host language for XPath like XML Schema, XSLT, Schematron, etc.

REFERENCES

AMER-YAHIA, S., CHO, S., LAKSHMANAN, L. V. S., AND SRIVASTAVA, D. 2001. Minimization of tree pattern queries. *SIGMOD Record 30,* 2, 497–508.

BARCENAS, E., GENEVÈS, P., LAYAÏDA, N., AND SCHMITT, A. 2011. Query reasoning on trees with types, interleaving and counting. In *IJCAI'11 : Proceedings of the 22nd International Joint Conference on Artificial Intelligence*. 718–723.

BENEDIKT, M., FAN, W., AND GEERTS, F. 2008. XPath satisfiability in the presence of DTDs. *J. ACM 55,* 2, 8:1–8:79.

BENZAKEN, V., CASTAGNA, G., COLAZZO, D., AND NGUYÊN, K. 2006. Type-based XML projection. In *Proceedings of the 32nd international conference on Very large data bases*. VLDB '06. VLDB Endowment, 271–282.

BERGLUND, A., BOAG, S., CHAMBERLIN, D., FERNÁNDEZ, M. F., KAY, M., ROBIE, J., AND SIMÉON, J. 2006. XML path language (XPath) 2.0, W3C candidate recommendation. http://www.w3.org/TR/xpath20/.

BOAG, S., CHAMBERLIN, D., FERNÁNDEZ, M. F., FLORESCU, D., ROBIE, J., AND SIMÉON, J. 2007. XQuery 1.0: An XML query language, W3C recommendation.

CHEKOL, M. W., EUZENAT, J., GENEVÈS, P., AND LAYAÏDA, N. 2012a. SPARQL query containment under RDFS entailment regime. In *IJCAR: Proceedings of the 6th International Joint Conference on Automated Reasoning*. 134–148.

CHEKOL, M. W., EUZENAT, J., GENEVÈS, P., AND LAYAÏDA, N. 2012b. SPARQL query containment under SHI axioms. In *AAAI: Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*.

CLARK, J. AND DEROSE, S. 1999. XML path language (XPath) version 1.0, W3C recommendation. http://www.w3.org/TR/ 1999/REC-xpath-19991116.

CURINO, C., MOON, H. J., DEUTSCH, A., AND ZANIOLO, C. 2013. Automating the database schema evolution process. *The VLDB Journal 22,* 1, 73–98.

CURINO, C. A., MOON, H. J., AND ZANIOLO, C. 2008. Graceful database schema evolution: the prism workbench. *Proc. VLDB Endow. 1,* 1, 761–772.

FALLSIDE, D. C. AND WALMSLEY, P. 2004. XML Schema part 0: Primer second edition, W3C recommendation. http://www.w3.org/TR/xmlschema-0/.

FOURNY, G., KOSSMANN, D., KRASKA, T., PILMAN, M., AND FLORESCU, D. 2008. XQuery in the browser. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. SIGMOD '08. ACM, New York, NY, USA, 1337–1340.

GENEVÈS, P. AND LAYAÏDA, N. 2006. A system for the static analysis of XPath. *ACM Transactions on Information Systems (TOIS) 24,* 4, 475–502.

GENEVÈS, P. AND LAYAÏDA, N. 2010. Eliminating dead-code from XQuery programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ICSE '10. ACM, New York, NY, USA, 305–306.

GENEVÈS, P. AND LAYAÏDA, N. 2010. XML reasoning made practical. In *ICDE'10: Proceedings of the 26th International Conference on Data Engineering*. IEEE, 1169–1172.

GENEVÈS, P. AND LAYAÏDA, N. 2011. Inconsistent path detection for XML IDEs. In *Proceeding of the 33rd international conference on Software engineering*. ICSE '11. ACM, New York, NY, USA, 983–985.

GENEVÈS, P. AND LAYAÏDA, N. 2011. Video demo. http://www.youtube.com/watch?v=uCFpxTEjj7g.

GENEVÈS, P., LAYAÏDA, N., AND QUINT, V. 2009. Identifying query incompatibilities with evolving XML schemas. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*. ICFP '09. ACM, New York, NY, USA, 221–230.

GENEVÈS, P., LAYAÏDA, N., AND QUINT, V. 2011. Impact of XML schema evolution. *ACM Trans. Internet Technol. 11*, 4:1–4:27.

GENEVÈS, P., LAYAÏDA, N., AND QUINT, V. 2012. On the analysis of cascading style sheets. In *WWW'12: Proceedings of the 21st World Wide Web Conference*. 809–818.

GENEVÈS, P., LAYAÏDA, N., AND SCHMITT, A. 2007. Efficient static analysis of XML paths and types. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '07. ACM, New York, NY, USA, 342–351.

GENEVÈS, P., LAYAÏDA, N., AND SCHMITT, A. 2008. Efficient static analysis of XML paths and types (extended version). Research Report 6590, INRIA. July.

GENEVÈS, P. AND VION-DURY, J.-Y. 2004. Logic-based XPath optimization. In *DocEng'04: Proceedings of the 2004 ACM Symposium on Document Engineering*. ACM Press, NY, USA, 211–219.

GESBERT, N., GENEVÈS, P., AND LAYAÏDA, N. 2011. Parametric polymorphism and semantic subtyping: the logical connection. In *ICFP '11: Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*. 107–116.

HOPCROFT, J. E., MOTWANI, R., ROTWANI, AND ULLMAN, J. D. 2000. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

HOSCHKA, P. 1998. Synchronized multimedia integration language (SMIL) 1.0 specification, W3C recommendation. http://www.w3.org/TR/REC-smil/.

MARIAN, A. AND SIMÉON, J. 2003. Projecting XML documents. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*. VLDB '03. VLDB Endowment, 213–224.

MARX, M. 2004. Conditional XPath, the first order complete XPath dialect. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. PODS '04. ACM, New York, NY, USA, 13–22.

MIKLAU, G. AND SUCIU, D. 2004. Containment and equivalence for a fragment of XPath. *Journal of the ACM 51,* 1, 2–45.

MURATA, M., LEE, D., MANI, M., AND KAWAGUCHI, K. 2005. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology 5,* 4, 660–704.

TERMEHCHY, A., WINSLETT, M., CHODPATHUMWAN, Y., AND GIBBONS, A. 2012. Design independent query interfaces. *IEEE Trans. Knowl. Data Eng. 24,* 10, 1819–1832.

TRUONG, B. Q., BHOWMICK, S. S., AND DYRESON, C. 2012. Sinbad: towards structure-independent querying of common neighbors in xml databases. In *Proceedings of the 17th international conference on Database Systems for Advanced Applications - Volume Part I*. DASFAA'12. Springer-Verlag, Berlin, Heidelberg, 156–171.

VEILLARD, D. 2013. Libxml2. http://www.xmlsoft.org/.

WADLER, P. 2000. Two semantics for XPath. Internal Technical Note of the W3C XSL Working Group, http://homepages.inf.ed.ac.uk/wadler/papers/xpath-semantics/xpath-semantics.pdf.

XALAN. 2013. Apache foundation Xalan-C++. http://xml.apache.org/xalan-c/.

XQDT. 2011. XQDT : XQuery development tools. http://wiki.eclipse.org/XQDT.