Type-based Static and Dynamic Website Verification

Jorge Coelho
Instituto Superior de
Engenharia do Porto & LIACC
jcoelho@ncc.up.pt

Mário Florido
University of Porto,
DCC-FC & LIACC
amf@ncc.up.pt

Abstract— **Maintaining large websites and verifying their semantic content is a difficult task. In this paper we propose a framework for syntactic validation, semantic verification and automatic correction of websites based on the logic programming language XCentric. Here we purpose a new approach conciliating the highly declarative model of XCentric with compile and run time verification techniques, mainly based on type checking to automatically repair and audit websites. The result is an easy to follow model to improve and audit website content.**

I. Introduction

Maintainance of a large website can be a tedious and difficult task. If the website is built by many different people the problem increases considerably. Now, suppose we want to impose certain type of restrictions on the content provided in this website. For example, imagine you administer a faculty website and you want to guarantee that all the teachers provide information on their personal home pages about the courses they teach and about their scientific research. There is a also lot of information that can be used to infer more information. For example, in our department we have all the publications added to a central page and we would like to infer some statistical data from it. Executing these tasks manually is tedious. In this paper we present a framework for syntactic validation and semantic verification of content which extends our previous work (VeriFlog [1]) with the ability to automatically repair the web pages that don't obey to a given rule and to verify if the changes introduced do not violate any of the rules provided for the website. We accomplish these tasks by introducing type declarations and compile time type checking along with consistency verification and type-checking during run-time.

Our base language is XCentric [2]. This language achieves a high level of expressivity through a new unification model based on flexible arity function symbols and sequence variables. We recently purposed in [3] the extension of the language with types, providing a mean of checking, querying and processing in a simpler way. XCentric provides a very pleasant way to query data in html/xml documents and to write verification rules. The framework works by translating documents to terms (an internal representation for documents), and then optionally applying syntactic validation, verifying semantics and inferring new data. All the different modules are implemented in XCentric which is built on top of SWI-Prolog [4], thus the programmer can use all the potential of SWI-Prolog in addition to XCentric.

Let's now present a simple example of use of the framework. Let's suppose we have a XML file describing a list of publications which is valid against the following DTD:

```
<!ELEMENT bib(pub*)>
<!ELEMENT pub (author+, title, booktitle, volume?,
                                    year?, publisher?)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT booktitle (#PCDATA)>
<!ELEMENT volume (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
```

and we want to impose one simple constraint:

- If the year of publication is greater than 2006, the string "to appear" must occur in the "booktitle" content. If it doesn't then simply add it.

This can be done by the following program:

```
swc(URLPub,NewPub):-
    xml2pro(URLPub,XML),
    bind_type(XML,type_pub),
    r1(XML1,XML2),
    pro2xml(XML2,NewPub).

r1(X1,X2):-
 replace(pub(X,booktitle(BT1),Y),
                 pub(X,booktitle(BT2),Y),X1,X2,
                [Y =*= <_,year(Y1),_>,
                 atom2number(Y1,Yn),
                 Yn > 2006, not(substring("to appear",BT1)),
                 concat(BT1,"(to appear)",BT2]).
```

In this program we retrieve an XML document from a url given in *URLPub*, then it is converted to the internal representation by builtin *xml2pro* and binded to type *type_pub* which represents the DTD presented before. The rule is described in predicate *r1*, where *pub* is used as a flexible arity functor (thus X and Y may have zero or more elements) and for all publications where the year is greater than 2006 and the string "to appear" does not occur in the booktitle, the string "to appear" is automatically added to the content of booktitle. This is done for all publications on $X1$ which violate the set of constraints resulting in a new document $X2$. During compile time a static analysis is carried on in order to see if the rule violates the declared type. During run-time the types are again used in order to verify that the values introduced don't change the document in wrong way. In case there is more than one rule, a consistency check is also made in order to insure that actions made by one rule does not violate other rule.

In the rest of the paper we assume that the reader is familiar with logic programming ([5]) and XML ([6]). We start in section 2 by presenting related work, then in section 3 we present the concepts behind XCentric language, we proceed in

section 4 by introducing types and in section 5 the incomplete terms in depth. In section 6 we present our new verification framework along with some examples, finally we present future work and conclude.

II. Related Work

Website verification was addressed in several previous works. In [7] the authors present a rewriting-based framework that uses simulation [8] in order to query terms, this was a new rewriting-based language quite different from ours. In [9], the authors present a semi-automatic methodology for repairing faulty websites by applying a set of concepts from Integrity Constraint [10] thus, very different from our type-based approach wich uses static checking and run time validation. In [11] the author proposed the use of a simple pattern-matching-based language and its translation to Prolog as a framework for website verification. Our work also uses Prolog but our syntax smoothly integrates with it, thus our framework inherits all the power of Prolog. We also provide a richer interface to semistructured data. In [12] logic was proposed as the rule language for semantic verification, there the authors provide a mean for introducing rules in a graphical format. In contrast, our work provides a powerful programming language and thus a richer and more flexible way to write rules. In [13] the author proposed an algorithm for website verification similar to [14] in expressiveness but based in a different theoretical approach. The idea was to extend sequence and non-sequence variable pattern matching with context variables, allowing a more flexible way to process semistructured data but the author doesn't provide an implementation.

III. XCentric

Here we present briefly the concept behind XCentric along the lines presented in [15].

A. Constraint Logic Programming

Constraint Logic Programming (CLP) [16] is the programming paradigm used in a class of languages based on rule-based constraint programming. Each different language is obtained by specifying the domain of discourse and the functions and relations on the particular domain. This framework extends the logic programming framework because it extends the Herbrand universe, the notion of *unification* and the notion of equation, accordingly to the new computational domains. A complete description of the major trends of the fundamental concepts about CLP can be found in [16].

B. XCentric

XCentric extends Prolog with terms with flexible arity symbols and sequence variables. We now describe the syntax of XCentric programs and their intuitive semantics.

In XCentric we extend the domain of discourse of Prolog (trees over uninterpreted functors) with finite sequences of trees.

Definition 1: A *sequence* $\tilde{t}$, is defined as follows:

- ε is the empty sequence.
- $t_1, \tilde{t}$ is a sequence if t_1 is a term and $\tilde{t}$ is a sequence

Example 1: Given the terms $f(a)$, b and X, then $\tilde{t} = f(a), b, X$ is a sequence.

Equality is the only relation between trees. Equality between trees is defined in the standard way: two trees are equal if and only if their root functor are the same and their corresponding subtrees, if any, are equal.

We now proceed with the syntactic formalization of XCentric, by extending the standard notion of Prolog term with flexible arity function symbols and sequence variables.

We consider an alphabet consisting of the following sets: the set of standard variables, the set of sequence variables (variables are denoted by upper case letters), the set of constants (denoted by lower case letters), the set of fixed arity function symbols and the set of flexible arity function symbols.

Definition 2: The set of terms over the previous alphabet is the smallest set that satisfies the following conditions:

1) Constants, standard variables and sequence variables are terms.
2) If f is a flexible arity function symbol and $t_1, \ldots, t_n$ ($n \geq 0$) are terms, then $f(t_1, \ldots, t_n)$ is a term.
3) If f is a fixed arity function symbol with arity n, $n \geq 0$ and $t_1, \ldots, t_n$ are terms such that for all $1 \leq i \leq n$, t_i does not contain sequence variables as subterms, then $f(t_1, \ldots, t_n)$ is a term.

Terms of the form $f(t_1, \ldots, t_n)$ where f is a function symbol and $t_1, \ldots, t_n$ are terms are called *compound terms*.

Definition 3: If t_1 and t_2 are terms then $t_1 = t_2$ (standard Prolog unification) and $t_1 = * = t_2$ (unification of terms with flexible arity symbols) are constraints.

A constraint $t_1 = * = t_2$ or $t_1 = t_2$ is solvable if and only if there is an assignment of sequences or ground terms, respectively, to variables therein such that the constraint evaluates to *true*, i.e. such that after that assignment the terms become equal.

Remark 1: In what follows, to avoid further formality, we shall assume that the domain of interpretation of variables is predetermined by the context where they occur. Variables occurring in a constraint of the form $t_1 = * = t_2$ are interpreted in the domain of sequences of trees, otherwise they are standard Prolog variables. In XCentric programs, therefore, each predicate symbol, functor and variable is used in a consistent way with respect to its domain of interpretation. XCentric programs have a syntax similar to Prolog extended with the new constraint $= * =$. The operational model of XCentric is the same of Prolog.

C. Constraint Solving

Constraints of the form $t_1 = * = t_2$ are solved by a non-standard unification that calculates the corresponding minimal complete set of unifiers. Details about the implementation of this non-standard unification can be found in [2]. As motivation we present some examples of unification:

Example 2: Given the terms $f(X, b, Y)$ and $f(a, b, b, b)$ where X and Y are sequence variables, $f(X, b, Y) = * = f(a, b, b, b)$ gives three results:

1) $X = a$ and $Y = b, b$
2) $X = a, b$ and $Y = b$
3) $X = a, b, b$ and $Y = \varepsilon$

Example 3: Given the terms $f(b, X)$ and $f(Y, d)$ where X and Y are sequence variables, $f(b, X) = * = f(Y, d)$ gives two possible solutions:

1) $X = d$ and $Y = b$
2) $X = N, d$ and $Y = b, N$ where N is a new sequence variable.

Note that this non-standard unification is conservative with respect to standard unification: in the last example the first solution corresponds to the use of standard unification. Soundness and completeness of this non-standard unification were proved in [17] and [15].

D. XML Processing

In XCentric an XML document is translated to a term with flexible arity function symbol. This term has a main functor (the root tag) and zero or more arguments. Although our actual implementation translates attributes to a list of pairs, since attributes do not play a relevant role in this work we will omit them in the examples, for the sake of simplicity. Consider the simple XML file presented bellow:

```
<addressbook>
        <record>
                <name>John</name>
                <address>New York</address>
                <email>john.ny@mailserver.com</email>
        </record>
        ...
</addressbook>
```

The equivalent term is:

```
addressbook(record(name('John'),address('New York'),
                   email('john.ny@mailserver.com')),...)
```

Example 4: Suppose that the term Doc is the XCentric representation of the document "addressbook.xml". If we want to gather the names of the people living in New York we can simply solve the following constraint:

$$\text{Doc} = * =$$
$$\text{addressbook}(\_, record(\text{name(N)}, \text{address('New York')}, \_), \_).$$

All the solutions can then be found by backtracking.
Note that '_' is an unnamed sequence variable which unifies with any sequence. Further details and examples can be found in [2], [15].

IV. Types

In this section we present the type language starting with a brief description of *Regular Types* and then their extension to type sequences of terms along the lines presented in [3] and [18].

A. Regular Types

The next definitions and examples introduce briefly the notion of Regular Types along the lines presented in [19].

Definition 4: Assuming an infinite set of *type symbols*, a *type term* is defined as follows:

1) A constant symbol (we use $a, b, c, etc.$) is a type term.
2) A type symbol (we use $\alpha, \beta, etc.$) is a type term.
3) If f is a flexible arity function symbol and each τ_i is a type term, $f(\tau_1, ..., \tau_n)$ is a type term.

Definition 5: A *type rule* is an expression of the form $\alpha \rightarrow \Upsilon$ where α is a type symbol and Υ is a finite set of type terms.

Example 5: Let α and β be type symbols, $\alpha \rightarrow \{a, b\}$ and $\beta \rightarrow \{nil, tree(\beta, \alpha, \beta)\}$ are type rules.

Definition 6: A type symbol α is defined by a set of type rules T if there exists a type rule $\alpha \rightarrow \Upsilon \in T$.
We make some assumptions:

1) The set of constant symbols are partitioned in non-empty subsets, called *base types*. Some examples are, *string*, *int*, and *float*.
2) The existence of μ, the universal type, and ϕ representing the empty type.
3) Each type symbol occurring in a set of type rules T is either μ, ϕ, a base type symbol, or a type symbol defined in T, and each type symbol defined in T has exactly one defining rule in T.

Regular types are the class of types that can be defined by finite sets of type rules. In XCentric a type rule $\alpha \rightarrow \{\tau_1, \ldots, \tau_n\}$ is represented by the declaration:

$$\text{:-type } \alpha \text{ - - -> } \tau_1; \ldots; \tau_n.$$

B. Regular Expression Types

We now define regular expression types, which describe sequences of values: a* (sequence of zero or more a's), a+ (sequence of one or more a's), a? (zero or one a), a|b (a or b) and a,b (a followed by b).

We use a special kind of terms, here called *sequence terms*, for implementing sequences.

Definition 7: A *sequence term*, $\bar{t}$ is defined as follows:

- ϵ is a *sequence term* that represents the empty sequence.
- $seq(t, \bar{s})$ is a *sequence term* if t is a term and $\bar{s}$ is a *sequence term*.

Definition 8: A sequence term in *normal form* is defined as:

- ϵ is in normal form
- $seq(t_1, t_2)$ is in normal form if t_1 is not of the form $seq(t_3, t_4)$ and t_2 is in normal form.

Example 6: Given the function symbol f, the variable X and the constants a and b: $seq(f(\ seq(a,\ \epsilon)),\ seq(b,\ seq(X,\ \epsilon)))$ is a *sequence term* in normal form.
Sequence terms in normal form are the internal representation of sequences. For example, $seq(a, seq(b, \epsilon))$ represents sequence a,b. Note that for simplification purposes we drop the seq operators for sequences of just one element.

We translate regular expression types to our internal sequence notation:

$$\begin{array}{lcl} a* & \Rightarrow & \alpha_* \rightarrow \{\epsilon, seq(a, \alpha_*)\} \\ a+ & \Rightarrow & \alpha_+ \rightarrow \{a, seq(a, \alpha_+)\} \\ a? & \Rightarrow & \alpha_? \rightarrow \{\epsilon,\ seq(a, \epsilon)\} \\ a|b & \Rightarrow & \alpha_| \rightarrow \{a, b\} \\ a, b & \Rightarrow & \alpha_{seq} \rightarrow \{seq(a, seq(b, \epsilon))\} \end{array}$$

Note that DTDs (Document Type Definition) [6] can be trivially translated to regular expression types.

Example 7: The DTD,

```
<!ELEMENT a (#PCDATA)>
<!ELEMENT b (c+)>
<!ELEMENT c (#PCDATA)>
<!ELEMENT d (#PCDATA)>
<!ELEMENT e (#PCDATA)>
<!ELEMENT l (a*, b, e?, d)>
```

corresponds to the regular expression type:

$\alpha \rightarrow l(a(string)*, b(c(string)+), e(string)?, d(string))$

XCentric also has some *XML Schema* [20] support:

- *Basic types*: string, integer, float and boolean.
- *Occurrences of sequences:* $\tau_{\{min,max\}}$, meaning a sequence of elements of type τ where the length of the sequence is between min and max. The example presented on the introduction, for the type of a sequence of two or more authors, can be written as:

  ```
  type type_a ---> author(string){2,unbounded}.
  ```

  Type $type\_a$ represents every sequence of two or more authors.
- *Orderless sequences:* $\{\tau_1 \& \tau_2 \& \ldots \& \tau_n\}$ meaning a sequence of elements of types $\tau_1, \tau_2, \ldots, \tau_n$ that can occur in any order. Consider the following type:

  ```
  type mix ---> rec({name(string) &
                     address(string) &
                         email(string)}).
  ```

  Type *mix* represents a record with three elements where their order doesn't matter.

Example 8: The following declaration introduces regular expression types describing terms in a simple bibliographic database:

```
:-type bib ---> bib(book+).
:-type book ---> book(author+, name).
:-type author ---> author(string).
:-type name ---> name(string).
```

Example 9: The next type describes arbitrary sequences of authors with at least two authors:

```
:-type type_a ---> (author(string),author(string),
                                  author(string)*).
```

C. Types as programs

It is well known that regular types can be associated with unary logic programs (see [21]–[24]). For every type symbol α, there is a predicate definition α, such that $\alpha(t)$ is true if and only if t is a term with type α (note that we are using the type symbol as the predicate symbol). The universal type μ is defined by the predicate $\mu(X) \leftarrow$. For every other type symbol, α, where $\alpha \rightarrow \Upsilon$ and $\tau \in \Upsilon$, the program has a clause,

$$\alpha(t) \leftarrow \beta(X_1) \wedge \ldots \wedge \beta(X_n)$$

where t is τ with each type symbol β_i replaced by a new variable x_i.

Example 10: Let α_{list} be a type symbol with type rule, $\alpha_{list} \rightarrow \{nil, .(a, \alpha_{list})\}$. α_{list} can be associated with the program:

$$\begin{aligned} \alpha_{list}(nil) &\leftarrow \\ \alpha_{list}(.(a, X)) &\leftarrow \alpha_{list}(X) \end{aligned}$$

Let $\alpha_1, \ldots, \alpha_n$ be type symbols defined by the rules in T. Φ_T denotes the program that defines μ (universal type) and $\alpha_1, \ldots, \alpha_n$.

Example 11: Let $T = \{\alpha \rightarrow \{\epsilon, a\}, \beta \rightarrow \{seq(\alpha, \gamma)\}, \gamma \rightarrow \{\epsilon, seq(d, \gamma)\}\}$ then, Φ_T is:

$$\begin{aligned} \alpha(\epsilon) &\leftarrow \\ \alpha(a) &\leftarrow \\ \beta(seq(X_1, X_2)) &\leftarrow \alpha(X_1), \gamma(X_2) \\ \gamma(\epsilon) &\leftarrow \\ \gamma(seq(d, X_2)) &\leftarrow \gamma(X_2) \end{aligned}$$

The *type* associated with a type symbol α in a set of type rules T is the set of terms occurring as arguments to the unary predicate α in the minimal model M_{Φ_T} (details about standard semantics of logic programs can be found in [5]).

Definition 9: Let T be a set of type rules. The *type* associated with the pure type term τ with respect to T is given by the following recursive definition (the first line applies when τ is constant symbol c, the second one applies when τ is type symbol α and the third one when τ is the pure type term $f(\tau_1, \ldots, \tau_n)$):

$$[\tau]_T = \begin{cases} \{c\} \\ \{t | \alpha(t) \in M_{\Phi_T}\} \\ \{f(t_1, \ldots, t_n) | t_i \in [\tau_i]_T, 1 \leq i \leq n\} \end{cases}$$

Types that can be described by $[\tau]_T$, where T is a set of type rules are called *regular types*. Informally, $[\tau]_T$ is the set of terms that can be derived from τ by repeated application of rules in T.

Example 12: Let T be the set of type rules $\{\alpha \rightarrow \{\epsilon, seq(a, \alpha)\}\}$, then $[\alpha]_T = \{\epsilon, seq(a, \epsilon), seq(a, seq(a, \epsilon)), \ldots\}$

V. Incomplete terms in depth

We provide predicates that allow the programmer to find a sequence of elements at arbitrary depth, to search for the nth occurrence of a sequence of elements and to count the number of occurrences of a sequence. The predicates are *deep/2*, *deepp/3* and *deepc/3*, respectively.

For example, consider we have an XML document bound to the variable *XML*. Suppose we want to find a sequence of elements between two elements named *incision* and store it in variable *Critical*. We can do the query:

```
? - deep(<incision(_),Critical,incision(_)>,XML).
```

If we want to find the text of the third occurrence of element author in document Bib we can simply write:

```
? - deepp(author(T),Bib,3).
```

Here, variable T will be instantiated with the name of the author. If we want to count the number of occurrences of $author$ elements in document $Book$ we can simply do:

```
<report>
  <section>
    <section_title>Procedure</section_title>
     <section_content>
      The patient was taken to the operating room where
      she was placed...
     <anesthesia>induced under general anesthesia.
      </anesthesia>
     <prep>
       <action>A Foley catheter was placed to decompress
       </action>
       and the abdomen was then prepped and draped in
       sterile fashion.
     </prep>
     <incision>
       A curvilinear incision was made
       <geography>in the midline immediately
                  infraumbilical</geography>
       and the subcutaneous tissue was divided
       <instrument>using electrocautery.</instrument>
     </incision>
     The fascia was identified and
     <action>#2 0 Maxon stay sutures were placed on
              each side of the...
     </action>
     <incision>
       The fascia was divided using
       <instrument>electrocautery</instrument>
       and the peritoneum was entered.
     </incision>
     <observation>
         The small bowel was identified.
     </observation>
     and <action> the <instrument>
                  Hasson trocar</instrument>
      was placed under direct visualization.
     </action>
     <action>
       The <instrument>trocar</instrument>
       was secured to the fascia using the stay sutures.
     </action>
    </section_content>
  </section>
</report>
```

Fig. 1. report1.xml

```
? - deepc(author(_),Book,C).
```

Note that all these predicates use sequences of elements.

Example 13: This example is based on a medical report using the HL7 Patient Record Architecture and inspired by the XQuery use cases available at [25]. Given *report1.xml* (Fig. 1), find what happened between the first *incision* and the second *incision* and write the result in a file named critical.xml:

```
translate:-
    xml2pro('report1.xml',Rep),
    deep(<incision(_),Critical,incision(_)>,Rep),
    newdoc(critical_sequence,Critical,FL),
    pro2xml(FL,'critical.xml').
```

The result is:

```
<critical_sequence>
 The fascia was identified and<action> #2 0
 Maxon stay sutures were placed on each side
 of the midline.</action>
</critical_sequence>
```

VI. Verification Framework

In this section we describe how the framework works. We start by defining a set of rules as the constraints for a given website. We introduce special builtins for dealing with errors and finally show how the static-time verification and the run-time consistency checking work. In this paper we focus on syntactic validation, semantic verification and error correction. The framework keeps the same tools for querying and inferring data as presented in [1] and thus we address the interested reader to [1] for further details on this specific topic. The framework can be used as a tool for the webmaster which feeds it with a set of pages to verify and correct them.

Another approach can be the use of the framework as an auditing tool for everyone which publishes in the website. The webmaster describes the rules and the system is used to verify the content of any page prior to its publication online. The page can only be published if the system confirms it is correct accordingly to the imposed constraints.

A. Website Constraints

Definition 10: We define a set of website constraints (SWC) as follows:

- A main rule whose input is a web page ($WpageI$) and output is a new web page ($WpageO$) resulting from the input page with the necessary changes in order to obey to the set of constraints imposed:
$$swc(WpageI, WpageO) \quad :- \quad r_1(WpageI, Wpage1), \quad \vdots, \quad r_n(Wpage_{n-1}, WpageO).$$
- Each rule r_i imposes some action to be taken in case some set of constraints is violated. We call these kind of rule an *action rule*. The rule may change the original document in order to make it obey to the constraint set.

Note that $WpageI$ and $WpageO$ is our internal representation for XML data and thus $WpageI$ resulted from translating a web page from a URL or file by using one of the internal builtins available in the framework.

We now define three new builtins associated with action rules:

- $delete(S, WpageI, WpageO, L)$ - deletes sequence S which respect the constraints in L from $WpageI$ resulting in $WpageO$.
- $replace(S_1, S_2, WpageI, WpageO, L)$ - replace sequence S_1 which respect the constraints in L by sequence S_2 in $WpageI$ by $WpageO$.
- $failure(WebpageI, L, Mesg)$ - used when the error is to serious to be automatically solved. Message $Mesg$ is shown when the constraints in L are violated.

Elements in L are, for example, tests in the values present in the sequences in order to verify they follow a certain criteria.

Example 14: Given the following web page:

```
<teacher>
   <name>Mario</name>
   <phone>+351 123456789</phone>
   <email>amf@ncc.up.pt</email>
   <teaching>
     <course>Compilers</course>
     <course>Theory of Computation</course>
   </teaching>
</teacher>
```

which is translated to the internal representation (stored in variable W_1):

```
teacher(
        name(''Mario''),
        phone(''+351 123456789''),
        email(''amf@ncc.up.pt''),
        teaching(course(''Compilers''))
                 course(''Theory of Computation''))).
```

If we apply:

$$delete(< phone(\_), email(\_) >, W_1, W_2, []).$$

W_1 will be translated to:

```
teacher(
        name(''Mario''),
        teaching(course(''Compilers''),
                 course(''Theory of Computation''))).
```

where the *phone* and *email* tags have been deleted. If for example we want to delete all the names of course which do not occur in our database in variable DB, one can do:

$$delete(course(N), W_1, W_2, [not(deep(course(N), DB))]).$$

If for example "Theory of Computation" does not occur in DB the result is:

```
teacher(
        name(''Mario''),
        teaching(course(''Compilers''))).
```

Example 15: Given the following XML file for a catalog of books translated to a term in W_1:

```
<catalog>
  ...
  <book number="500">
    <name>
     Haskell: The Craft of Functional Programming
     (2nd Edition)
    </name>
    <author>Simon Thompson</author>
    <price>41</price>
    <year>1999</year>
  </book>
  <book number="501">
    <name>
      Data on the Web
    </name>
     <author>Serge Abiteboul</author>
     <author>Peter Buneman</author>
     <author>Dan Suciu</author>
    <price>null</price>
    <year>2000</year>
  </book>
  ...
</catalog>
```

to replace all the prices with non-numeric values by 0 one can do:

$$replace(price(X), price(0), W_1, W_2, [not(number(X))]).$$

To reduce 10% to all the prices higher than 45 one can do:

$$replace(price(X), price(Y), W_1, W_2, [X > 45, Y = X - (X * 0.1)]).$$

We now proceed with an example of running sets of website constraints.

Example 16: Let's use again a teacher's webpage which is valid with respect to the following DTD:

```
<!ELEMENT teacher (name, phone, email*, research,
                   teaching, curriculum)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT research (pub*)>
<!ELEMENT pub (author+, title, booktitle, volume?,
               year?, publisher?)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT booktitle (#PCDATA)>
<!ELEMENT volume (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT teaching (course+)>
<!ELEMENT course (#PCDATA)>
<!ELEMENT curriculum (#PCDATA)>
```

This DTD can be trivially translated to the following regular expression type (and included in the program file):

```
:-type teacher ---> teacher(name(string), phone(string),
                            email(string)*, research,
                            teaching,curriculum).
:-type name ---> name(string).
:-type phone ---> phone(string).
:-type email ---> email(string).
:-type research ---> research(pub*).
:-type pub ---> pub (author+, title, booktitle,
                     volume?, year?, publisher?).
:-type author ---> author(string).
:-type title ---> title(string).
:-type booktitle ---> booktitle(string).
:-type volume ---> volume(string).
:-type year ---> year(string).
:-type publisher ---> publisher(string).
:-type teaching ---> teaching (course+).
:-type course ---> course(string).
:-type curriculum ---> curriculum(string).
```

We want to verify the following constraints:

- The teacher must have a non empty curriculum tag in his homepage and this tag should include a small text were the teacher resumes his curriculum: he should include a reference to the degree and the year he obtained it. In case this rule is not obeyed, a small warning text should appear in the curriculum tag.
- A well-formed email address should be available, otherwise the verification should be canceled and the error reported.
- All the publications the teacher has that don't appear in the central repository should be deleted.

We can build the program presented in Fig. 2 to verify the above constraints over the teacher's webpage. It works as follows: the url is retrieved and stored in variable $WpageI$ using builtin *xml2pro*. Then, *bind_type(Page,Type)* binds the webpage *Page* to the previous declared type *Type*. Without *bind_type* the compiler ignores the type information and thus would not be possible to find errors at compile time. The rules are described in the remaining lines of the program.

This example and many others are available at:

http://www.ncc.up.pt/∼jcoelho/veriflog/

B. Static verification of action rules

Static verification is made by static analysis of action rules. These rules possibly change the document by means of a delete or replace, thus having type information describing the document structure, it is possible to verify the validity of these transformations.

Definition 11: We define the type of a term representing an XML document as:

```
start:-
        xml2pro('./examples/amf.xml',Tea),
        xml2pro('./examples/pubs.xml',Pub),
        assert(pubs(Pub)),
        bind_type(Tea,teacher),
        swc(Tea,Tea2),
        pro2xml(Tea2,'teacherchecked.xml'),
        retract(pubs(_)).


swc(In,Out):-
        r1(In,Out1),
        r2(Out1,Out2),
        r3(Out2,Out).

r1(A,B):-
     replace(curriculum(X),
             curriculum('Degree and year missing!'),A,B,
             [deep(curriculum(X),A),
              not(sub_string(X,'degree')),
              not(sub_string(X,'year'))]).

r2(A,A):-
     failure(A,[deep(email(X),A),
                not(sub_string(X,'@ncc.up.pt'))],
                'Valid email not found!').

r3(A,B):-
        delete(pub(X),A,B,[deep(pub(X),A),
                           X =*= <_,title(T),_>,
                           pubs(Pub),
                           not(deep(title(T),Pub))]).
```

Fig. 2. Teacher's home page constraints

$$type(X) = pcdata, \text{ if } X \text{ is not of de form } f(s_1,\ldots,s_n) \text{ where } n \geq 1$$
$$type(f(s_1,\ldots,s_n)) = f(type(s_1),\ldots,type(s_n))$$

Example 17: Given the following term:

$$teacher(name("Jorge"), phone("123456789")),$$

then,

$$type(teacher(name("Jorge"), phone("123456789"))) = teacher(name(pcdata), email(pcdata)),$$

During compile time the type verification procedure follows these steps:

- A type α is associated with document XML by a $bind\_type$ instruction.
- For any instruction of the form $delete(s_1, XML, XML2, L)$:
  - Verify if s_1*, $s_1?$ or $s_{1_{\{0,Max\}}}$ is found in the DTD, in case it isn't, report an error.
  - If $L \neq []$ and s_1+ or $s_{1_{N,Max}}$ where $N > 0$ was found then report a warning saying that if the constraints in L covers all the elements in the document, then $XML2$ will be an invalid document.
- For any instruction of the form $replace(s_1, s_2, XML, XML2, L)$, if $type(s_1) \neq type(s_2)$:
  - If s_1 and s_2 appear as a sequence in the type:
    - Report an error in cases were s_1 and s_2 appear as:
      - s_1, s_2 (because s_1 and s_2 must occur exactly once).
      - s_1, s_2*; s_1, s_2+; $s_1, s_2?$; $s_1, s_{2_{\{N',M'\}}}$, and $L = []$ (because s_1 must occur once).
      - s_1+, s_2; s_1*, s_2; $s_1?, s_2$; $s_{1_{\{N,M\}}}, s_2$; $s_{1_{\{N,M\}}}, s_2?$, if $N \neq 0$ and $M \neq 1$ (because only one s_2 is allowed).
      - s_1+, s_2+; s_1+, s_2*; $s_1+, s_{2_{\{N,M\}}}$; $s_1+, s_2?$, and $L = []$, (because at least one s_1 must occur).
      - $s_{1_{\{N,M\}}}, s_2+$; $s_{1_{\{N,M\}}}, s_{2_{\{N',M'\}}}$;$s_{1_{\{N,M\}}}, s_2*$: if $N \neq 0$ and $L = []$ (because s_1 must occur N times at least).
    - Report a warning in cases were s_1 and s_2 occur as:
      - $s_1, s_2?$; $s_{1_{\{0,1\}}}, s_2?$: *s_2 must not occur in the document.*
      - s_1+, s_2+; s_1+, s_2* and $L \neq []$ *the constraints in L cannot replace all $s'_1 s$.*
      - $s_1+, s_2?$; $s_1*, s_2?$ and $L \neq []$ *at most one s_1 can be translated and s_2 must not occur.*
      - $s_1*, s_{2_{\{N',M'\}}}$; $s_1?, s_{2_{\{N',M'\}}}$, *the number of $s'_1 s$ plus the number of $s'_2 s$ may not exceed M'.*
      - $s_1?, s_2?$: *s_2 must not occur.*
      - $s_{1_{\{N,M\}}}, s_2+$; $s_{1_{\{N,M\}}}, s_2*$, and $L \neq []$ *the number of $s'_1 s$ which remain not replaced must be higher than N.*
  - If s_1 and s_2 occur as a disjunction (|) in the type:
    - Report an error in cases were s_1 and s_2 occur as:
      - $s_1?|s_{2_{\{N',M'\}}}$ and $N' \geq 2$ (because only one s_1 will be translated).
      - $s_{1_{\{N,M\}}}|s_2$; $s_{1_{\{N,M\}}}|s_2?$, and $N' \geq 1$ (because only one s_2 is allowed).
      - $s_{1_{\{N,M\}}}, s_{2_{\{N',M'\}}}$, if $[N,M] \cap [N',M'] = \emptyset$ (because they will never have a compatible number of occurences).
    - Report a warning in cases were s_1 and s_2 occur as:
      - $s_1 + |s_2$; $s_1 + |s_2?$; $s_1 * |s_2$; $s_1 * |s_2?$: *because only one s_2 is allowed.*
      - $s_1 + |s_{2_{\{N,M\}}}$; $s_1 * |s_{2_{\{N,M\}}}$, *because $N \leq$ number of $s'_1 s \leq M$.*
      - $s_{1_{\{0,M\}}}|s_2+$, *at least one s_1 must be translated.*
      - $s_{1_{\{N,M\}}}|s_{2_{\{N',M'\}}}$, $[N,M] \cap [N',M'] \neq \emptyset$ *because $N' \leq$ number of $s'_1 s \leq M'$.*

Example 18: Using the same scenario as the one presented in example 16. Errors are reported if for example we try to apply some of the following actions:

- $delete(name(X), XML_1, XML_2, [])$ since $name$ is a tag that cannot be deleted from the document.
- $replace(phone(P), email("test@testmail.com"), XML_1, XML_2, [])$ since althought we can have several emails, the phone element is mandatory.

Warnings are reported if we try the following action:

- $delete(author(A), XML_1, XML_2, [internal\_database(DB), not(DB = *= db(\_, name(A), \_))])$ here the goal is to delete all the authors whose name does not appear in another XML file in variable DB. A warning is produced warning that, if all the authors for a given

record are covered, then the XML produced will be invalid.

Note that this verification were all made at compile time.

C. Run time consistency checking

Types declared for the documents are transformed in programs as explained in section IV-C and used for checking the content of the document given as input and the document generated as output.

Create a new program from the original SWC which, after applying the verification, is executed in order to find any possible inconsistency in the set of constraints.

Given the following SWC:

```
swc(WpageI,WpageO):-
        r_1(WpageI,Wpage1),
        ...
        r_n(Wpage_{n-1},WpageO).

r_1(W1,W2):-...
...
r_n(W1,W2):-...
```

a new program is generated where the body of each rule is replaced by the constraints found inside each action and an error message. Then, after applying the original SWC, check the output XML file using this new program. This way it is possible to detect inconsistency errors between the rules.

Example 19: Using the scenario presented in example 16, the generated program for consistency errors detection is:

```
verify(WpageO):-
            r1(WpageO),
            r2(WpageO),
            r3(WpageO).

r1(W):-
    deep(curriculum(X),W),
    not(sub_string(X,'degree')),
    not(sub_string(X,'year')),
    write("Rule 1 fails!"),!.

r1(_).


r2(W):-
    deep(email(X),W),
    not(sub_string(X,'@ncc.up.pt')),
    write("Rule 2 fails!"),!.

r2(_).

r3(W):-
    deep(pub(X),W),
    X =*= <_,title(T),_>,
    pubs(Pub),
    not(deep(title(T),Pub)),
    write("Rule 3 fails!"),!.

r3(_).
```

Example 20: Suppose we add to the teacher's SWC presented in example 16 the following constraint:

```
r4(W1,W2):-
    delete(email(E),W1,W2,[]).
```

Although not detected at compile time, since not having an email is valid accordingly to the DTD, this error would be detected when checking the rules at run-time by:

```
r2(W):-
    deep(email(X),W),
    not(sub_string(X,'@ncc.up.pt')),
    write("Rule 2 fails!"),!,fail.
```

And thus rule number 4 made an inconsistent change in relation to rule number 2.

Note that the output XML document is again checked against the declared types. This way we can find errors from actions which depend on values.

VII. Conclusion

In this paper we present a framework for website verification at compile-time and run-time using type verification of rules applicable to documents. Errors can be automatically solved by introducing actions that are executed whenever an error is found. Errors within the set of web constraints imposed by the system administrator are detected by type checking at compile time complemented with further type checking at run time and consistency analysis to detect constraints whose action may violate other constraints. XML-based websites although not yet widespread, will probably become more and more popular in the next years thus increasing the utility of a tools such as the one presented in this paper.

Acknowledgment

Research presented in this paper has been partially supported by funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *Programa POSI*.

References

[1] J. Coelho and M. Florido, "VeriFLog: Constraint Logic Programming Applied to Verification of Website Content," in *Int. Workshop XML Research and Applications (XRA'06)*, ser. LNCS, vol. 3842. Springer-Verlag, 2006.

[2] ——, "Clp(flex): Constraint logic programming applied to xml processing," in *Ontologies, Databases and Applications of SEmantics (ODBASE)*, ser. LNCS, vol. 3291. Springer Verlag, 2004.

[3] ——, "Unification with flexible arity symbols: a typed approach," in *Informal proceedings of the 20th International Workshop on Unification (UNIF'06)*, Seattle, USA, 2006.

[4] S. Prolog, http://www.swi-prolog.org/.

[5] J. W. Lloyd, *Foundations of Logic Programming*, 2nd ed. Springer-Verlag, 1987.

[6] E. M. L. (XML), http://www.w3.org/XML/, 2003.

[7] M. Alpuente, D. Ballis, and M. Falaschi, "A Rewriting-based Framework for Web Sites Verification," in *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2005, pp. 41–61.

[8] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke, "Computing simulations on finite and infinite graphs." in *FOCS*, 1995, pp. 453–462.

[9] M. Alpuente, D. Ballis, M. Falaschi, and D. Romero, "A semi-automatic methodology for repairing faulty web sites," in *WWV*, M. Alpuente, S. Escobar, and M. Falaschi, Eds. Departamento de Sistemas Informaticos y Computacion, Universidad Politecnica de Valencia, 2005.

[10] E. Mayol and E. Teniente, "A survey of current methods for integrity constraint maintenance and view updating," in *ER '99: Proceedings of the Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling*. London, UK: Springer-Verlag, 1999, pp. 62–73.

[11] T. Despeyroux, "Practical semantic analysis of web sites and documents." in *WWW*, S. I. Feldman, M. Uretsky, M. Najork, and C. E. Wills, Eds. ACM, 2004, pp. 685–693.

[12] F. van Harmelen and J. van der Meer, "Webmaster: Knowledge-based verification of web-pages." in *IEA/AIE*, ser. Lecture Notes in Computer Science, I. F. Imam, Y. Kodratoff, A. El-Dessouki, and M. Ali, Eds., vol. 1611. Springer, 1999, pp. 256–265.

[13] T. Kutsia, "Context sequence matching for xml," in *Proceedings of the 1th Int. Workshop on Automated Specification and Verification of Web Sites*, 2005.

[14] F. Bry and S. Schaffert, "Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification," in *International Conference on Logic Programming (ICLP)*, ser. LNCS, vol. 2401, 2002.
[15] J. Coelho and M. Florido, "CLP(Flex): Constraint logic programming applied to XML processing," DCC-FC, LIACC. University of Porto, (available from www.ncc.up.pt/~jcoelho/clpflex.pdf), Tech. Rep. 06, July 2004.
[16] J. Jaffar and M. J. Maher, "Constraint logic programming: A survey," *Journal of Logic Programming*, vol. 19/20, pp. 503–581, 1994.
[17] T. Kutsia, "Unification with sequence variables and flexible arity symbols and its extension with pattern-terms." in *Joint AISC'2002 - Calculemus'2002 conference*, ser. LNAI, 2002.
[18] J. Coelho and M. Florido, "Xcentric: A logic programming language for xml," DCC-FC, LIACC. University of Porto, (available from http://www.ncc.up.pt/xcentric/xcentric.pdf), Technical Report, 2006.
[19] P. Dart and J. Zobel, "A regular type language for logic programs," in *Types in Logic Programming*, F. Pfenning, Ed. The MIT Press, 1992.
[20] X. Schema, http://www.w3.org/XML/Schema/, 2000.
[21] J. Zobel, "Derivation of polymorphic types for prolog programs," in *Proc. of the 1987 International Conference on Logic Programming*. MIT Press, 1987.
[22] E. Yardeni and E. Shapiro, "A type system for logic programs," in *The Journal of Logic Programming*, 1990.
[23] M. Florido and L. Damas, "Types as theories," in *Proc. of post-conference workshop on Proofs and Types, Joint International Conference and Symposium on Logic Programming*, 1992.
[24] T. W. Fruhwirth, E. Y. Shapiro, M. Y. Vardi, and E. Yardeni, "Logic programs as types for logic programs," in *LICS*, 1991, pp. 300–309.
[25] XQuery Use Cases, http://www.w3.org/TR/xquery-use-cases/, 2005.