Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 235 (2009) 137–152
www.elsevier.com/locate/entcs

Web Sites Repairing through Abduction

Paolo Mancarella, Giacomo Terreni[1]

Dipartimento di Informatica
Universitá di Pisa
Pisa, Italy

Francesca Toni[2]

Department of Computing
Imperial College
London, UK

Abstract

We present a methodology and a tool for suggesting repairs to web sites that violate some given requirements in the form of web rules expressed in (an extension of) a fragment of Excerpt. The methodology consists in translating these web rules into abductive logic programs with constraints and process these by means of an existing general-purpose proof procedure, called CIFF. The tool, that we call CIFFWEB, consists of CIFF as well as the translation from rules to programs and from web sites to a suitable logical format. The tool extends an existing tool for simply checking web sites against web rules.

Keywords: Abduction, Web site repairing

1 Introduction

The exponential growth of the WWW raises the question of maintaining and repairing automatically web sites, in particular when the designers of these sites require them to exhibit certain properties at both structural and data level. The capability of maintaining and repairing web sites is also important to ensure the success of the Semantic Web [3] vision. As this relies upon the definition and the maintenance of consistent data schemas (XML/XMLSchema, RDF/RDFSchema, OWL and many other formal languages [3]), the web needs to reason with such schemas.

We strongly believe that declarative languages such as Logic Programming (LP) [9], if integrated with the web, will play a crucial role as computational paradigms in the Semantic Web vision, as noted, e.g., in [15]. Also abduction [14], as it is a

[1] Email: paolo@di.unipi.it, terreni@di.unipi.it
[2] Email: ft@doc.ic.ac.uk

doi:10.1016/j.entcs.2009.03.009

very suitable form of reasoning for diagnosis and repairing, could play a preminent role in the Semantic Web vision, as noted, e.g., in [7].

In this paper, we propose the CIFFWEB tool, a prototype tool for checking and suggesting repairs of XML web sites against sets of requirements which have to be fulfilled by a web site instance. CIFFWEB uses through abductive reasoning in LP as realised by the CIFF proof procedure [11].

In [13,14], we define the *web checking rules*, i.e. an expressive characterization of rules for checking web sites' errors by using (a fragment of) the well-known semi-structured data query language Xcerpt [6]. Then, we mapped them into *programs for checking*, i.e. abductive logic programs with constraints that can be fed as input to the general-purpose CIFF abductive proof procedure [11,14].

We deploy the CIFF System, a Prolog implementation of the CIFF proof procedure, to reason upon the (translation of the) web checking rules finding those XML/XHTML instances not fulfilling the rules, and representing errors as abducibles in abductive logic programs. By mapping web checking rules onto abductive logic programs with constraints and deploying CIFF for determining fulfillment (or identify violation) of the rules, we inherit the soundness properties of CIFF thus obtaining a sound concrete tool for web checking.

In this paper, we extend our earlier work for *suggesting* possible repairing actions for web site instances containing errors. In this respect, abducibles may represent not only error instances fired by an XML/XHTML instance violating a rule r (for checking) but also possible modifications (repairs) to that XML/XHTML data such that both r is fulfilled and no other rules are violated.

We identify some types of errors, arising from the violation of web checking rules, which are suitable to be abductively repaired, and we define a further mapping from web checking rules to another type of abductive logic programs with constraints: *programs for repairing*. Again, through the use of programs for repairing with CIFF, for determining fulfillment of the rules, or suggesting appropriate repairing actions, we inherit the soundness properties of CIFF thus obtaining a sound concrete tool for web repairing.

2 A Motivating Example

Searching the web, it is easy to encounter web pages containing errors in their structure and/or their data. We argue that, in most cases, considering an XML/XHTML web site instance, the errors can be divided into two main categories: structural errors and content-related (data) errors. Structural errors are those errors concerning the presence and/or absence of tag elements and relations amongst tag elements in the pages. For example, if a tag $tag1$ is intended to be child of a tag $tag2$, the occurrence in the web site of a $tag1$ instance outside the scope of a $tag2$ instance is a structural error. Data errors, instead, are about the in-tag data contents of tag elements. For example a $tag3$ could be required to hold a number greater than 100. To better exemplify the types of error we consider, we present here a very simple

XML web page representing a list of shows produced by a theater company. [3]

```
%%%showindex.xml
<showlist>
  <show>                              <show>
    <showname>Mela</showname>           <showname>Epiloghi</showname>
    <year>1998</year>                   </show>
  </show>                             </showlist>
```

We could specify a number of rules which any web site for shows should fulfill. For example, we could specify that the right *structure* of a `show` tag must admit a `showname` tag element and a `year` tag element as its children. In the example, we would have a *structural error*, due to the lack of a `year` tag element in the second show. Moreover we could specify that `showlist` must contain only those `shows` produced since the year `2000`. In this case we would have a *data error* due to the first `show` being produced in `1998`.

Requirements (and thus errors) can involve more than one web page. Here, we omit this feature for lack of space (but see [13,14] for a discussion of this feature).

In the example in this section, we have assumed that an error instance is fired by a piece of XML/XHTML data which does not fulfill a certain requirement (or specification). We will first formalize any such requirement as a *web rule* that can be used for checking and repairing web sites through suitable uses of CIFF, resulting in the CIFFWEB system.

3 Background: checking web sites

In this section we briefly present the checking capabilities of CIFFWEB, fully addressed in [13,14]. We start with some background notions about web rules and abductive logic programming with constraints. Then we give a very brief description of the translation process from web checking rules to abductive logic programs for checking (*programs for checking* for short) and finally we show a CIFFWEB run on the theater example.

3.1 Web rules

In order to formalize and characterize requirements, such as the ones expressed in natural language in the earlier section, as web checking rules, we first need a formal language. Our choice is to use the Xcerpt [6] language: a deductive, rule-based query language for semi-structured data which allows for direct access to XML data in a very natural way. Our characterization of web checking rules can be accommodated straightforwardly in (an extension of) a fragment of the Xcerpt language. Here, we give some background notions about the Xcerpt fragment we use [4]. An Xcerpt program is composed of a `GOAL` part (*error part* in the sequel)

[3] Throughout the paper, we use the convention that each code-line starting with `%` is a line of comment.

[4] The full Xcerpt language is much more expressive then the fragment we adopt here for expressing web checking rules. For further information about Xcerpt see [6].

and a FROM part (*condition part* in the sequel). The FROM part provides access to the sources (XML files or other sources) via (partial) pattern matching among terms, while the GOAL part reassembles the results of the query into new terms. Variables can be used within either parts and act as placeholders (as in logic programming). As an example, the requirement, in the context of our earlier example, that *the showlist must contain shows produced since year 2000 [Rule1]* can be expressed as:

```
%%%%%%%Rule 1 - Show produced before year 2000
GOAL  all err [ var Year, "show produced before 2000" ]
FROM
       in {
        resource {"file:showindex.xml"},
          year {{ var Year }}
       }
       where ( var Year < 2000)
END
```

The main Xcerpt statement we use in the GOAL part is the `all` *e* statement (where *e* is a term, `err` in our example), indicating that each possible XML instance satisfying the FROM part gives rise to a new instance of *e* returned by the GOAL part. In our methodology for writing web checking rules, `all` *e* will always be `all err`, where `err` stands for "error".

In the FROM part, an access to a `resource` is wrapped within an `in` statement that also includes a query term *q*. Accesses to multiple pages must be connected by `and` indicating that all queries have to succeed in order to make the whole query succeed. The main Xcerpt components we use in our queries are: (1) double curly brackets, i.e. *q*{{ }}, denoting *partial term specification* of *q*; the order of the subterms of *q* within the curly brackets is irrelevant; (2) variables, expressed by `var` followed by an identifier (variable name); values for variables can be strings and numeric values; (3) a `where` statement for expressing constraints through standard operators like `=, \=, <, >, <=`; (4) subterms of the form `without` *s* denoting *subterm negation*, illustrated within the following formulation of the requirement that *each show must have a production year [Rule2]*:

```
%%%%%%%Rule 2 - No year tag inside a show tag
GOAL all err [ "show without a year tag" ]
FROM
       in {
        resource {"file:showindex.xml"},
          show {{
            without year {{ }}
          }}
       }
END
```

`without` is only applicable to subterms *s* that do not occur at *root level* in the underlying web pages (in our example, `showlist` occur at root level); a `without` subterm

cannot occur nested within another `without` subterm and finally all variables that occur within a `without` have to appear elsewhere outside the `without` (we will refer to the last constraint as to the "Xcerpt allowedness").

In order to accommodate the absence of *any* raw data inside a tag, we define an extension of Xcerpt, using the `without_data` statement as an Xcerpt query subterm. This is a very useful statement because one of the most common errors in a web site instance is the absence of data inside a tag which cannot be expressed by the standard `without` due to the Xcerpt allowedness. As an example we express the requirement that *each year tag must contain some data [Rule3]*

```
%%%%%%%Rule 3 - No data inside a year tag
GOAL all err [ "year tag without data" ]
FROM
       in {
        resource {"file:showindex.xml"},
          year {{ without_data }}
       }
END
```

This rule would be violated, for example, deleting the `1998` value in the `year` tag of the example. In the sequel, we denote a web rule as a *negative rule* if it contains either a `without` or a `without_data` statement and *positive rule* otherwise. The full grammar of web rules, omitted here for lack of space, can be found in [13,14].

3.2 Abductive logic programming with constraints

An *abductive logic program with constraints* (ALPC) consists of (1) a constraint logic program P, referred to as the *theory*, namely a set of clauses of the form $A \leftarrow L_1 \wedge \ldots \wedge L_m$, where the L_is are literals (ordinary or abducible atoms, their negation, or constraint atoms in some underlying language for constraints), A is an ordinary atom, and all the variables in the clause are all implicitly universally quantified from the outside; (2) a finite set of *abducible predicates*, that do not occur in any conclusion A of any clause in the theory, and (3) a finite set of *integrity constraints* (ICs), namely implications of the form $L_1 \wedge \cdots \wedge L_m \rightarrow A_1 \vee \cdots \vee A_n$ where the L_is are literals (the *body* of the IC) and the A_js are (ordinary, abducible, constraint or $false$) atoms (the *head* of the IC), and whose variables are all implicitly universally quantified from the outside. The theory provides definitions for ordinary predicates, while constraint atoms are evaluated within an underlying structure $\Re$, as in conventional constraint logic programming. Abducibles can be used to extend the theory, subject to satisfying the ICs.

A *query* is a conjunction of literals (whose variables are implicitly existentially quantified). An *answer* to a query specifies which instances of the abducible predicates should hold so that both (1) (some instance of) the query is entailed by the constraint logic program extended with the abducibles and (2) the ICs are satisfied [11]. More precisely, chosen the three-valued completion semantics [12] as the underlying logic programming semantics, let $\models_{3(\Re)}$ denote the notion of entailment in the

3-valued completion appropriately augmented, à-la-constraint logic programming, with a notion of satisfiability for the underlying structure $\Re$ for interpreting the constraint atoms in the abductive logic program. Then, an *answer* for a query Q is a triple $\langle \Delta, \sigma, \Gamma \rangle$ where Δ is a set of abducible atoms, σ is a substitution for the variables appearing in Q, and Γ is a set of constraints atoms such that there exists a ground substitution σ' for the variables occurring in $Q\sigma \cup \Delta \cup \Gamma$ such that $\sigma'' = \sigma\sigma'$ and $P \cup \Delta\sigma'' \models_{3(\Re)} Q\sigma'' \wedge IC$.

The CIFF proof procedure computes such answers. It operates with a presentation of the theory as a set of *iff-definitions*, which are obtained by the (selective) *completion* of all predicates defined in the theory except for the abducible and the constraint predicates. CIFF returns three possible outputs: (1) a computed answer to the query (a set of possibly non-ground abducible atoms and a set of constraints on the variables of the query and of the abducible atoms); (2) a failure, indicating that there is no answer and (3) an *undefined* answer, indicating that part of the input is not *allowed*. Allowedness relates to input formulae with certain quantification patterns for which the concept of a (finite) answer cannot be defined [11].

Computed answers are abductive answers in the semantic sense.

3.3 From web rules to abductive logic programs for checking

To pave the way to writing, in a format suitable for the CIFF system, specification rules for properties of web sites, we first provide a suitable representation of web sites data. Obviously CIFF is not able to handle directly XML data and XML structure, hence we propose a translation whereby for each tag element of the original XML file, an atom `pg_el(ID,TagName,IDFather)` is part of the abductive logic program with constraints CIFF reasons with. Here `TagName` is the name of the tag element, whereas `ID` and `IDFather` represent the unique identifiers for the tag element and its father. They are needed for keeping information about the structure of the XML page. Similarly, raw data inside a tag is represented by a `data_el(ID,Data,IDFather)` atom. The following is the translation of the XML page seen in section 2:

```
xml_pg('showindex.xml',0).
pg_el(1,showlist,0).
pg_el(2,show,1).
pg_el(3,showname,2).
data_el(4,'Mela',3).
pg_el(5,year,2).
data_el(6,'1998',5).
pg_el(7,show,1).
pg_el(8,showname,7).
data_el(9,'Epiloghi',8).
```

For each page, we store also a fact of the form `xml_pg(FileName,ID)`, holding the name of the file and the unique identifier of the page and serving as the "root" of the elements. Note that CIFFWEB automatically translates the XML pages appearing in any `resource` statement of the involved web rules.

The structure of a web checking rule directly recalls the structure of a CIFF integrity constraint: in particular, the *condition part* can be mapped into the body of an integrity constraint I and the *error part* can be mapped into the head of I. This

is exactly how *positive rules* are mapped into abductive logic programs. The head of I is composed of an abducible predicate `abd_err/2` whose arguments are (1) the variable list occurring in the error part of the rule and (2) the error message. The body of I instead, is composed of the conjunction of the components occurring in the *condition part* of the rule, represented as appropriate `pg_el` and `data_el` atoms. The translation of *Rule1* is the following [5]:

```
%%%% Each show is not produced before 2000 %%%
[pg_el(ID1,year,_), data_el(ID2,Year,ID1), Year#<2000]
implies
[abd_err([Year],'show produced before 2000')].
```

The idea is that each instance of the XML data matching the body will fire an instance of the head representing the corresponding error. Being `abd_err` an abducible predicate, this will amount to abducing its head and, thus, returning the error instance as an abducible atom.

Note that in the translation, new `ID` variables are introduced in order to make the program aware of the XML structure.

In the example, the body is matched if in the XML representation there is an atom `data_el(ID2,Year,ID1)`, which contains a value for `Year` less than `#< 2000` and whose father `ID1` represents a `year` tag.

The mapping of *negative rules* is a bit more complicated: the resulting abductive logic program is no longer a single integrity constraint but it includes also a set of clauses defining new (fresh) predicates needed for handling correctly `without` and `without_data` statements. The translation of *Rule3* is the following:

```
[pg_el(ID1,show,_), not(pred_1(ID1))]
implies
[abd_err([],'show without a year tag')].

pred_1(ID1) :- pg_el(ID2,year,ID1).
```

Intuitively, the body is fired if there is no way to satisfy `pred_1`, i.e. there is no atom `pg_el(ID2,year,ID1)` in the XML representation whose father `ID1` is a `show` tag.

The `without_data` statement, instead, is handled by defining a `some_data(ID)` predicate which is satisfied if there is *any* `data_el(_,_,ID)` atom, whose father is `ID`, in the XML representation. The translation of *Rule3* and the definition of `some_data` are as follows:

```
[pg_el(ID1,year,_), not(some_data(ID1))]
implies
[abd_err([],'year tag without data')].

some_data(ID) :- data_el(_,_,ID).
```

[5] We directly use the concrete CIFF System syntax, where an IC of the form $L_1 \wedge \cdots \wedge L_m \rightarrow A_1 \vee \cdots \vee A_n$ is represented as `[L1, ..., Lm] implies [A1, ..., An]` and variable names start with a capital letter.

3.4 Running CIFFWEB for checking the example

In order to run the CIFFWEB system[6] the web rules in Xcerpt syntax and the XML/XHTML pages are needed. The query is always the empty query (also for repair). The system will compile all the rules and the sources, producing the corresponding program for checking. Running the system with the three rules and the XML page seen above, the following abductive answer is produced:

```
[abd_err(['1998'],'show produced before 2000'),
abd_err([],'show without a year tag'].
```

representing correctly that the show `'Mela'` is produced before the year `'2000'` and that the show `'Epiloghi'` has been inserted without a production year.

It is worth noticing that the checking task could also be represented in a deductive way, translating the web rules in normal logic programs, rather than in CIFF integrity constraints, and then querying the system for returning all the `abd_err` instances. However, with the abductive account proposed here we obtain a coherent framework when the repairing task, which is a typical abductive task, is taken into account.

4 A Web Repairing Framework

Abductive reasoning can also be exploited for repairing errors of a web site, translating web checking rules into more complex abductive logic programs. These abductive logic programs can be used with CIFF in order to *suggest*, through abductive answers, how to repair the errors.

There can be several error types in a web site which could be repaired in more than one way as noted in [2]. For example there can be duplicated data and a repairing action can be identified as a *deletion* action. Or there may be a wrong tag name or a wrong data item, e.g. a wrong result of a sum, in that case a suitable repairing action may be to *change* the tag name or the data which caused the error. A third error type is a missing data: in that case an *insert* action is arguably the best repair action. However, in many situations, it is difficult to choose the appropriate repair action. Consider two data tags $A1 = 500$ and $A2 = 400$ together with a tag $A3$ which is supposed to contain the sum of the values $A1$ and $A2$. Assume that the value of $A3$ is 1000: a changing action should be performed, but on which data? The sum or one of the addends? It is clear that it will be very difficult for an automatic tool to decide which data has to be changed. However, such an automatic tool could *suggest* to human experts some repair action and leave the decision to them.

In our proposal, we take into account the errors which could be repaired by *insert* actions, i.e. those errors which arise, arguably, from missing XML data. We have chosen to deal with this type of error because on the one hand abduction is very suitable to *insert* information in a "given world" and, on the other hand, in our framework for web checking there is a straightforward relation between web check-

[6] The CIFFWEB System is available at *www.di.unipi.it/∼terreni/research.php*

ing rules and errors arising from missing data: *negative rules* lead to errors of that type (we denote those errors as *negative errors* and each error drawn from a *positive rule* as a *positive error*). This is very intuitive because the `without` and the `without_data` statements express the absence of information. For example, each error arising from the negative rule *Rule2* seen before, represents the absence of a `year` tag as a child of a `show`. The idea is that an abductive answer could suggest to *insert* such a `year` tag in the XML data. We leave the repair of other error types as future work.

4.1 Abductive logic programs for repairing

In this section we show how we can generate abductive logic programs for repairing (*programs for repairing* for short), which can be used by an abductive reasoner for abducing missing XML elements, from correspondent *programs for checking*.
Generalizing what we have seen in Section 3, a generic *negative rule* is mapped into a program for checking of the form:

```
[el_1(ID1,X1,IDF1), ..., el_M(IDM,XM,IDFM),
not(pred_1(Arg1)), ..., not(pred_N(ArgN))] implies [abd_err(Arg)]

pred_1(Arg1) :-
   el_11(ID_11,X_11,IDF_11),...,el_1T1(ID_1T1,X_1T1,IDF_1T1)
...
pred_N(ArgN) :-
   el_N1(ID_N1,X_N1,IDF_N1),...,el_NT1(ID_NT1,X_NT1,IDF_NT1)
```

where each `el_i` is either a `pg_el` atom or a `data_el` atom. Given a certain instance of `el_1, ..., el_M` in the XML specification, an error is detected if the XML specification satisfies *none* of the correspondent `pred_1, ..., pred_N` instances. In the case of repair, the idea is that error could be "repaired" by *abducing* the XML elements which satisfy at least one of the `pred_i` instances. Intuitively, from the above program for checking, our goal is to generate a program for repairing where (1) the integrity constraint is of the form

```
[el_1(ID1,X1,IDF1), ..., el_M(IDM,XM,IDFM)] implies
[pred_1(Arg1), ..., pred_N(ArgN)]
```

and (2) the definition of each `pred_i` is such that XML elements could also be abduced and not only matched against the XML representation [7].
Hence, the first thing a program for repairing needs, is a way to abduce XML elements. Thus, we declare two new abducible predicates: i.e. the `abd_pg_el` and the `abd_data_el` predicates representing the "abducible versions" of `pg_el` and `data_el` respectively. These relationships are represented by the following clauses added to a program for repairing (similarly for the `data_el` case):

```
all_pg_el(ID,TagName,IDFather) :- pg_el(ID,TagName,IDFather)
```

[7] Recall that in the CIFF syntax the list in the body of an integrity constraint represents a *conjunction* while the list in the head represents a *disjunction*

```
all_pg_el(ID,TagName,IDFather) :- abd_pg_el(ID,TagName,IDFather)
```

The new `all_pg_el` and `all_data_el` predicates (`all_i` for short) could be used in the `pred_i` definitions, replacing the `el_i` predicates. In this way a `pred_i` instance is satisfied either abductively or not.

However, a program for repairing should not abduce indiscriminately XML elements, but only those elements for fixing a "real" lack of data. Considering again the theater example, no `year` tag should be abduced within the first `show` tag because it already contains a production `year` in the original XML data. The above modification does not prevent this behavior and to solve this issue we introduce new `rep_pred_i` predicates defined as follows:

```
rep_pred_i(Argi) :- pred_i(Argi)
rep_pred_i(Argi) :- not(pred_i(Argi)),
   all_i1(ID_i1,X_i1,IDF_i1),..., all_iT1(ID_iTi,X_iT1,IDF_iTi)

pred_i(Argi) :-
   el_i1(ID_i1,X_i1,IDF_i1),..., el_iT1(ID_iTi,X_iT1,IDF_iTi)
```

Intuitively a `rep_pred_i` predicate is satisfied either if the correspondent `pred_i` is satisfied as well (i.e. without abductions) or abducing at least a XML element if `pred_i` is not satisfied. Note that the presence of `not` in the second clause makes the clauses defining `rep_pred_i` mutually exclusive. As one can expect those `rep_pred_i` are put in the head of an integrity constraint in a *program for repairing*:

```
[el_1(ID1,X1,IDF1), ..., el_M(IDM,XM,IDFM)] implies
[rep_pred_1(Arg1), ..., rep_pred_N(ArgN)]
```

The abduction of new XML elements leads to another issue to be taken into account: new abducibles introduced to repair an error could violate another web checking rule. A program for repairing must be aware of this problem repairing or, at least, detecting these new errors. In order to make a program for repairing aware of the new abducibles, we simply replace the el_i elements by their all_i counterparts in the body of the integrity constraints:

```
[all_1(ID1,X1,IDF1), ..., all_M(IDM,XM,IDFM)] implies
[rep_pred_1(Arg1), ..., rep_pred_N(ArgN)]
```

In this way each abduced XML element can satisfy the body of an integrity constraint leading to new abductions for repairing a *chain* of errors. This modification must be applied also to the bodies of the integrity constraints drawn from positive rules: in that case *positive errors* due to abduced XML elements are simply detected as for positive errors detected in the original XML data. The program for repairing obtained by the three web checking rules of the theater example is the following:

```
%%%Rule1
[all_pg_el(ID1,year,_), all_data_el(ID2,Year,ID1), Year#<2000]
implies
[abd_err([Year],'show produced before 2000')].
```

```
%%%Rule2
[all_pg_el(ID1,show,_)] implies [rep_pred_1(ID1)].

pred_1(ID1) :- pg_el(ID2,year,ID1).
rep_pred_1(ID1) :- pred_1(ID1).
rep_pred_1(ID1) :- not(pred_1(ID1)), all_pg_el(ID2,year,ID1).

%%%Rule3
[all_pg_el(ID1,year,_)] implies [rep_some_data(ID1)].

rep_some_data(ID) :- data_el(_,_,ID)
rep_some_data(ID) :- abd_data_el(_,_,ID), not(data_el(_,_,ID))
```

The `rep_some_data` predicate represents, in a *program for repairing*, the counterpart of the `some_data` predicate in a *program for checking*. Now, its definition takes into account the possible abduction of XML data.

In a *program for repairing* there is a last issue to cope with: the identifiers of the abduced XML elements. As seen in Section 3.3, each element in the original XML data is associated to a unique numerical identifier and these identifiers maintain the original XML tree structure. When a XML element is abduced a unique identifier should be assigned to it in a similar way. Obviously, for each pair of abduced XML elements, their identifiers should be distinct and each newly generated identifier should be distinct from each identifier of the original XML elements. This can be done by adding, in a *program for repairing*, a set of integrity constraints of the form:

```
[pg_el(X,Y1,Z1),abd_pg_el(X,Y2,Z2)] implies [false]
...
[data_el(X,Y1,Z1),abd_data_el(X,Y2,Z2)] implies [false]

[abd_pg_el(X1,Y1,Z1),abd_pg_el(X2,Y2,Z2),Y1 #\= Y2] implies [X1 #\= X2]
...
[abd_data_el(X1,Y1,Z1),abd_data_el(X2,Y2,Z2),Z1 #\= Z2] implies
       [X1 #\= X2]
```

The newly generated identifiers can be instantiated to appropriate numerical values when answers are extracted from CIF computed answers.

5 Running the CIFFWEB System for repairing

Running the CIFFWEB system with the program for repairing seen in the previous section we obtain the following abductive answer [8]:

```
[abd_err(['1998'],'show produced before 2000'),
```

[8] The concrete programs for repairing obtained through the automatic translations of web checking rules are an optimized version of the ones seen here. IDs are instantiated automatically by the CIFFWEB system. Further information can be found in [14]

```
 abd_pg_el(10,year,7),
 abd_data_el(11,X,10), X#>=2000].
```

Note that the error about the production year of the first show is drawn for a *positive rule* (*Rule1*) and the system behaves as for checking. The error of the missing `year` tag in the second show (drawn from *Rule2*), instead, is repaired by `abd_pg_el(10,year,7)` abducible which, correctly, represents a `year` tag inside the second `show` whose identifier is `7` in the XML representation. Note also that the identifier chosen by the system for the new abducible is `10` which does not clash with other identifiers. That abducible, however, produces a violation to *Rule3* because the new `year` tag does not contain data. Hence a further abduction is needed resulting to `abd_data_el(11,X,10)` where `X` is constrained to be greater or equal than `2000` due, again, to *Rule1*.
It is worth noticing that the behavior of the system is not straightforward due to the handling of errors chains and non-ground values during the computation.

6 Abductively Generated Errors

With the program for repairing presented so far for the example, the CIFFWEB system produces a further abductive answer:

```
[abd_err(['1998'],'show produced before 2000'),
 abd_err(['X'],'show produced before 2000'),
 abd_pg_el(10,year,7),
 abd_data_el(11,X,10), X#<2000].
```

Here, a further *positive error* `abd_err(['X'],'show produced before 2000')` is generated, with `X#<2000`. Namely, the system detects the need for adding a production year for the second show, but it suggests a "wrong" repair value. Thus the new error is detected. Note that from a declarative point of view, this is a sound solution to the problem. However it is less intuitive than the other computed answer.
The above case happens because the new abducibles match the body of *Rule1*, i.e.:

```
[all_pg_el(ID1,year,_), all_data_el(ID2,Year,ID1), Year#<2000]
implies [abd_err([Year],'show produced before 2000')].
```

Then if the `Year` variable (in the example the `X`), is constrained to be less than `2000` a new error is abduced too. This is general issue and we argue, that, usually, it might be preferable to avoid that abduced XML elements introduce new errors satisfying the bodies of the positive web checking rules. A way to avoid this is imposing that an instance of an integrity constraint obtained from a *positive rule* leads to a failure in the abductive process if its body is satisfied through (at least) an abduced atom. I.e. we could replace the abductive errors in the head of the integrity constraint by `false`. In this way, if an abduced atom leads to a new error, then the abductive process fails searching for an alternative abductive answer.
Consider an integrity constraint I, in a program for repairing, which represents a positive web checking rule, i.e.:

```
[all_1(ID1,X1,IDF1), ..., all_M(IDM,XM,IDFM)] implies [abd_err(Arg)]
```

Suppose now to "unfold" all the `all_i` atoms. What we obtain is a set of 2^M integrity constraints of the form:

```
[el_1(ID1,X1,IDF1),el_2(ID2,X2,IDF2),...,el_M(IDM,XM,IDFM)]
   implies [abd_err(Arg)]

[abd_1(ID1,X1,IDF1),el_2(ID2,X2,IDF2),...,el_M(IDM,XM,IDFM)]
   implies [abd_err(Arg)]
 ...
[abd_1(ID1,X1,IDF1),abd_2(ID2,X2,IDF2),...,abd_M(IDM,XM,IDFM)]
   implies [abd_err(Arg)]
```

where each `abd_i` represents the abducible version of the correspondent `el_i` atom. It is worth noticing that among the 2^m integrity constraints, only one of them does not contain abducibles in its body. For simplicity we say that this integrity constraint is I^1. If we want to avoid that the abduced atoms could not generate new errors we simply need to replace `abd_err` by `false` in each integrity constraint other than I^1. Adopting this solution, the latter abductive answer would not be returned by the system, because constraining `X` to a value less than `2000` would lead immediately to failure (since CIFF never computes answers containing `false`).
These two levels of *error generation* are both available in the CIFFWEB system by setting the `error_level` flag to `negative` (if no *positive errors* can be generated through abduced XML elements, the default value) or `any` otherwise. The system, automatically, preprocesses the integrity constraints as described above.

7 Analysis

Our translation provides a semantics for the fragment of Xcerpt we have adopted for defining web rules, and CIFFWEB a sound mechanism for performing the repairing process, by virtue of the soundness of CIFF for ALPCs.
Given a web site W, let $\mathcal{X}(W)$ be the set of ground unit clauses as the result of applying the translation illustrated in section 3.3 to W. Given a set of web checking rules R, let $Repair_R$ be the program for repairing obtained from R, i.e. an abductive logic program with constraints $\langle P, A, IC\rangle_{\Re}$ where A is the set { `abd_err, abd_pg_el, abd_data_el` } and P and IC are given as in Sections 4 and 6. Trivially, $\langle P \cup \mathcal{X}(W), A, IC\rangle_{\Re}$ is an ALPC.
By construction $IC = IC^- \cup IC^+$ where IC^- is the set of integrity constraints drawn from *negative rules* and IC^+ is the set of integrity constraints drawn from *positive rules*. Each $I^- \in IC^-$ is of the form:

```
[all_1(ID1,X1,IDF1), ..., all_M(IDM,XM,IDFM)] implies
[rep_pred_1(Arg1), ..., rep_pred_N(ArgN)]
```

while each integrity constraint in IC^+ is either an integrity constraint I^+ of the form:

`[atom_1(ID1,X1,IDF1),...,atom_M(IDM,XM,IDFM)] implies [abd_err(Arg)]`

or an integrity constraint I^+_{false} of the form:

`[atom_1(ID1,X1,IDF1),...,atom_M(IDM,XM,IDFM)] implies [false]`

where at least an `atom_i` is abducible. The presence of last type of integrity constraints depends on the chosen *error level*, as described in Section 6.

Consider an answer $\langle \Delta, \sigma, \Gamma \rangle$ for the empty query Q and with respect to $\langle P \cup \mathcal{X}(W), A, IC \rangle_{\Re}$. By definition, we have that there exists a ground $\sigma'' = \sigma\sigma'$ such that $P \cup \Delta\sigma'' \models_{3(\Re)} Q\sigma'' \wedge IC$.

Being Q empty, we only need to consider $P \cup \Delta\sigma'' \models_{3(\Re)} IC$.

By construction, we have that

$$\Delta = \Delta^{err} \cup \Delta^{el}$$

where Δ^{err} contains all the abduced `abd_err` atoms and Δ^{el} contains all the abduced `abd_pg_el` and `abd_data_el` atoms.

Let r^- be a negative rule and let I^- be the corresponding integrity constraint in IC^-. We say that r^- is fulfilled if for each ground instance b^- of the body of I^- such that

$$P \cup \mathcal{X}(W) \cup \Delta^{el}\sigma'' \models_{3(\Re)} b^- \qquad \text{then} \qquad P \cup \mathcal{X}(W) \cup \Delta^{el}\sigma'' \models_{3(\Re)} h^-$$

where h^- is the corresponding instance of the head of I^-.

Let r^+ be a positive rule and let I^+ and I^+_{false} the corresponding integrity constraints in IC^+. We say that r^+ is fulfilled (depending on the selected *error level*) either if for each instance ground instance b^+ of the body of I^+ such that

$$P \cup \mathcal{X}(W) \cup \Delta^{el}\sigma'' \models_{3(\Re)} b^+ \qquad \text{then} \qquad P \cup \mathcal{X}(W) \cup \Delta^{err}\sigma'' \models_{3(\Re)} h^+$$

where h^+ is the corresponding instance of the head pf I^+., or if there is no ground instance b^+_{false} of the body of I^+_{false} such that

$$P \cup \mathcal{X}(W) \cup \Delta^{el}\sigma'' \models_{3(\Re)} b^+_{false}.$$

By soundness of CIFF we obtain that, if CIFF for the empty query

- succeeds returning an empty answer then all rules in R are satisfied in W
- succeeds returning an answer with a non-empty Δ, then:
  - if Δ^{err} is non-empty then some positive rule in R is violated and
  - if Δ^{el} is non-empty then some error due to negative rules in R is repaired,
- returns no answer, then some integrity constraints of the form I^+_{false} are violated.

8 Conclusions

We have illustrated the CIFFWEB system for verifying and repairing XML/XHTML web sites by using abductive logic programming with constraints and in particular the CIFF proof procedure as computational counterpart. This work aims at helping coping with the exponential WWW growth and contributing to the success of the Semantic Web. There is limited work in the literature on verifying and repairing web sites at a semantic level. Notable exceptions, at least

for verifying web sites, are represented by [10]; the XLINKIT framework [8] and the GVERDI-R system [1,4].

The work more closely related to ours is the GVERDI-R system [1,4] which verifies web sites against correctness and completeness rules written in an ad-hoc language which relies upon a (partial) pattern-matching mechanism very similar to the Xcerpt one and whose expressiveness is comparable to our Xcerpt fragment. However, our use of negation constructs and our unordered subterm specification in the *condition part* of a rule allows for a bit more expressiveness. Another key difference is that the GVERDI-R system relies upon an ad-hoc computational counterpart for its framework, while our system relies upon the general purpose CIFF abductive proof procedure. Conversely, the GVERDI-R system allows for the use of functions for managing strings and data in a non-straightforward way, e.g. by matching strings to regular expressions or using arithmetic functions on numbers. While the CIFFWEB system deals with arithmetic functions thanks to the underlying integrated constraint solver, it lacks the use of other types of functions. As pointed out in [1], this is an important feature for a web verification tool. A more sophisticated use of functions in CIFFWEB is work in progress.

Another clear drawback of our system is the absence of a GUI, apart from the simple GUI for the Java translator, which allows for a better usability as for systems like GVERDI-R and XLINKIT.

We have shown a methodology for building programs for repairing web sites and then we have illustrated, by means of an example, how the CIFFWEB system is able, through the abductive computational core, to "suggest" non-straightforward and non-ground repairing actions in terms of abduced XML elements which, added to the original data, could repair missing XML data. To our knowledge, this is a novel feature which is absent in the other existing tools for web verification.

Also the GVERDI-R system seems to put some steps towards that direction as pointed out in [4], but a repairing specification and a concrete tool are still work in progress. In turn, the XLINKIT system seems to be capable of some form of repairing actions but they are limited to broken links in web sites.

Another interesting approach of modifying web data instances is represented by the XChange framework [5] which is proposed by the same Xcerpt authors and form which it derives. It proposes a framework to make the web data aware of events which should lead to changes in the web data. We are currently studying possible interrelations with our repair approach even if the Xchange framework is not focused to verify and to repair web site instances but rather it seems to propose new data paradigms which can accommodate event-driven changes.

References

[1] Alpuente, M., D. Ballis and M. Falaschi, *A rewriting-based framework for web sites verification*, Electronic Notes in Theoretical Computer Science **124** (2005), pp. 41–61.

[2] Alpuente, M., D. Ballis, M. Falaschi and D. Romero, *A semi-automatic methodology for repairing faulty web sites*, SEFM **0** (2006), pp. 31–40.

[3] Antoniou, G. and F. van Harmelen, "A Semantic Web Primer (Cooperative Information Systems)," The MIT Press, 2004.

[4] Ballis, D. and D. Romero, *Fixing web sites using correction stategies*, in: *Proc. of WWV'06*, 2006.

[5] Bry, F. and M. Eckert, *A high-level query language for events*, in: *Proc. of EDA-PS'06*, 2006, pp. 31–38.

[6] Bry, F. and S. Schaffert, *The XML query language Xcerpt: Design principles, examples, and semantics* (2002).

[7] C. Elsenbroich, O. K. and U. Sattler, *A case for abductive reasoning over ontologies*, in: *Proc. of OWL: Experiences and Directions*, 2006.

[8] Capra, L., W. Emmerich, A. Finkelstein and C. Nentwich, *XLINKIT: a consistency checking and smart link generation service*, ACM Transac. on IT **2** (2002), pp. 151–185.

[9] Colmerauer, A. and P. Roussel, *The birth of Prolog.*, in: *HOPL Preprints*, 1993, pp. 37–52.

[10] Despeyroux, T. and B. Trousse, *Semantic verification of web sites using natural semantics*, in: *Proc. of CC-BMIA'00*, 2000.

[11] Endriss, U., P. Mancarella, F. Sadri, G. Terreni and F. Toni, *The CIFF proof procedure for abductive logic programming with constraints*, in: *Proc. of JELIA 2004*, 2004.

[12] Kunen, K., *Negation in logic programming*, J. Log. Program. **4** (1987), pp. 289–308.

[13] Mancarella, P., G. Terreni and F. Toni, *Web sites verification: An abductive logic programming tool*, in: *Proc. of ICLP*, 2007, pp. 434–435.

[14] Terreni, G., "The CIFF Proof Procedure for Abductive Logic Programming with Constraints: Definition, Implementation and a Web Application," Ph.D. thesis, Università di Pisa (2008).

[15] Wielemaker, J., M. Hildebrand and J. van Ossenbruggen, *Using Prolog as the fundament for applications on the Semantic Web*, in: *Proc. of the 2nd Workshop on Applications of LP to the web, Semantic Web and Semantic Web Services*, 2007, pp. 84–98.