Reasoning About Web-Site Structure

Mary Fernandez
AT&T Research
mff@research.att.com

Daniela Florescu
INRIA Roquencourt
dana@rodin.inria.fr

Alon Levy
University of Washington
alon@cs.washington.edu

Dan Suciu
AT&T Research
suciu@research.att.com

Abstract

Building large Web sites is similar in many ways to building knowledge and database systems. In particular, by providing a declarative, *logical view* of a Web site's data and structure, many of a site builder's tasks, such as creating complex sites, modifying a site's structure, and creating multiple versions of a site, are simplified significantly. New systems, such as STRUDEL, support logical views of Web sites by allowing site builders to construct a site declaratively. In this paper, we address an important problem for site builders: verifying that a Web site's structure conforms to certain constraints. Specifically, we consider the problem of verifying that a Web site created declaratively by STRUDEL satisfies certain integrity constraints, such as 'all pages are reachable from the root' and 'every organization page points to its sub-organizations', etc. Our contributions are (1) formulating the verification problem as an entailment problem in a logical setting, and (2) presenting a sound and complete algorithm for verifying large classes of integrity constraints that occur in practice. Our algorithm uses a novel data structure, the *site schema*, which enables us to identify cases in which the general reasoning problem reduces to a decidable problem.

Introduction

The World-Wide Web (WWW) has given rise to a new form of knowledge base: the *Web site.* Web sites contain several bodies of data about the enterprise they are describing, and these bodies of data are linked into a rich structure. For example, a company's Web site may contain data about its employees, linked to data about the projects in which they participate and to the publications they author. The *data* presented at a Web site along with the *structure* of the links in the site together form a richly structured knowledge base.

The operations we wish to perform on Web sites are also often similar to those applied to knowledge bases. First, we want to inspect the information in the Web site. We can inspect the site by a combination of *querying* and *browsing.* We may inspect either the underlying data (e.g., find the price of a particular product), or query the site's structure to better focus our browsing (e.g., how do I find the homepage of a given person). Second, as builders of Web sites, we would like to *enforce constraints* on the structure of our site (e.g., no dangling pointers, an employee's homepage should point to their department's homepage). This problem is the focus of this paper. Third, we would like to be able to *easily modify* either the underlying data or the Web site's structure. Lastly, our ultimate goal is for our Web sites to be *adaptive* (Perkowitz and Etzioni 1997), e.g., we would like to *learn* from users' browsing patterns in order to improve a site's structure.

Although Web sites contain richly structured information, this structure is usually implicit in the Web site. In general, we do not have a *model* or *representation* of the site's structure and data. Some formalisms have been developed for providing post-hoc descriptions of Web sites (e.g., MCF (Guha 1997)). Even though such formalisms are useful for browsing sites, they do not facilitate modifications or updates. The above operations illustrate the possible benefits of viewing the problem of building Web sites from the perspective of building knowledge and data base systems. Allowing site builders to manipulate a *logical view* of the site, instead of individual HTML files, simplifies the construction and maintenance of Web sites. The logical view is the basis for services such as querying, enforcing constraints, and easy modification. In contrast, current Web site management tools provide only rudimentary support for such tasks.

STRUDEL (Fernandez *et al.* 1998) is a system for building Web sites starting from their *logical views.* The key idea is that Web sites are built by *declarative* specifications of the site's structure and content. In STRUDEL (see Figure 1) a Web site builder begins with a *data graph,* which is a model of the raw data to be presented at the site. For example, the data graph may model the personnel database and its contents, the set of publications, and images of employees. The site designer then specifies the Web site's structure in a declarative language called STRUQL. STRUQL describes a site's structure in a *lifted* (i.e., intensional) form, rather than in a *ground* form. For example, a STRUQL expression may contain a statement saying that *every* person has a homepage with their name and phone number, and that every person's homepage points to their department's homepage. Evaluating the STRUQL specification for a Web site on a given data graph results in a *site graph*, which is the ground specification of the site's structure. Intuitively, the site graph describes (1) what pages will be present at the Web site, (2) the information available in and the internal structure of each page, and (3) the links between pages. The STRUQL language has been designed such that Web sites can be constructed efficiently from their specifications. Formally, STRUQL corresponds to a restricted form of Horn rules, though, as we explain later, its syntax is appropriate for describing Web sites (and graphs in general). Finally, the Web site builder specifies a set of HTML *templates* that, when applied to the nodes in the site graph, result in an HTML page for each node, and hence to a browsable Web site. STRUDEL is a fully implemented system that has been used to build several medium-sized Web sites.

STRUDEL provides a platform for considering higher-level operations on Web sites, such as the ones described above. In this paper we consider one important problems in building Web sites: verifying constraints on the site's structure. Specifically, given a description of the Web site's structure in STRUQL, we want to check whether the resulting Web site is guaranteed to satisfy certain constraints (e.g., all pages are reachable from the root, every organization homepage points to the homepages of its suborganization, or proprietary data is not displayed on the external version of the site). It is tempting to think that because the structure of Web sites is specified declaratively, enforcing such constraints comes for free. In particular, why not specify the structure of the Web site and the constraints on its structure in the same declarative language (e.g., STRUQL)? The difference is that the specification of the structure generates a *unique* structure, while constraints are not generative, they only limit the set of possible structures. Hence, the challenge we face is to *reason* about whether the structure we have specified satisfies the required constraints. Furthermore, since specifications of complex Web sites require rather long STRUQL expressions, automating the reasoning task is important. Our work can be viewed as an instance of the knowledge-base verification problem, which has received significant attention (e.g., (Levy and Rousset 1996; Schmolze and Snyder 1997)) in the context of building Web sites.

The contributions of the paper are the following. We begin by presenting a formalization of the problem of verifying integrity constraints within a logical formalism. Intuitively, we formalize the problem as a question of logical entailment between two STRUQL expressions. We then consider the verification problem for a commonly occurring class of integrity constraints. Informally, this class of constraints specifies that certain kinds of paths *must* exist in the Web site. We provide a sound and complete algorithm for verifying that a STRUQL expression is guaranteed to yield a Web site that satisfies such a constraint. The key tool used in our algorithm is a novel data structure, the *site schema*, which represents a STRUQL expression as a labeled directed graph. Intuitively, this graph can be viewed as a *schema* of the Web sites that would result from the STRUQL expression. By analyzing the structure of the graph, we can write expressions that correspond to the possible paths in the Web site. Importantly, these expressions can be written in a language for which reasoning algorithms exist (a subset of datalog in one case, and a restricted form of STRUQL in another case). Hence, the analysis of the site schema yields algorithms for verifying the integrity constraints.

The focus of this paper is on the problem of verifying integrity constraints on Web sites. However, a broader contribution of this paper is to bring the problem of Web-site management to the attention of the Artificial Intelligence community. We argue that the declarative representation of Web sites given by Strudel provides a platform for exploring various issues in Web-site building and maintenance.

The Strudel System

In this section, we briefly describe the main components of STRUDEL's architecture (shown in Figure 1).

Overview

In STRUDEL, a site builder starts with *raw data,* then declaratively describes the content and structure of the site. The declarative description specifies (1) the pages in the site and the links between them, and (2) what raw data is displayed in each page. The raw data may exist in several external repositories, such as databases or structured files. Hence, STRUDEL has a *data integration* component (a.k.a. mediator) to provide the site builder a uniform view of all the data. This uniform view of the raw data is called the *data graph.*

A Web site's content and structure is specified in the STRUQL language, which we describe in detail below. As stated earlier, STRUQL is equivalent to a language that consists of a restricted form of Horn rules with function symbols. STRUQL's syntax, however, is quite different, because it was designed to (1) express queries over diverse sources of data such as databases (relational or object-oriented) and structured documents (e.g., a bibtex file), and (2) define explicitly the structure of graphs.

The STRUQL specification is a lifted description of a Web site's structure. Together with an instance of the data graph, the STRUQL specification uniquely defines the ground structure of the Web site, called the *site graph*. The site graph can be evaluated from the STRUQL specification and the data graph, much the same way a query is evaluated in a database system. We do not discuss the evaluation process in this paper, but note that STRUQL was designed to permit efficient evaluation.

Finally, we note that a site graph *does not* specify the graphical presentation of pages, therefore the last step when using STRUDEL is to define the graphical presentation of pages and generate the browsable Web site. The graphical presentation is specified by a set of *HTML templates*, which are HTML files with variables. Given a node in a site graph, an HTML template is instantiated by replacing variables in the template with the appropriate values from the node. Every node in a site graph has a corresponding HTML template, which may be unique to the node, but commonly is shared by a collection of related nodes. The browsable Web site is constructed by instantiating the appropriate HTML template for each node in the site graph.

STRUDEL's primary benefit is that it provides the Web-site builder a *logical view* of a site, instead of the physical view as a collection of statically linked HTML files. As a result, it is easier to (1) specify the structure of complex Web sites, (2) build different versions of a site (e.g., one version may be internal to a company, while another may be external), and (3) modify a site's structure and update its content. In this paper, we explore another benefit of building Web sites declaratively: specifying and verifying constraints on a Web site's structure. First, we describe STRUDEL's data model and define formally the STRUQL language.

Modeling Data in Strudel

STRUDEL's conceptualization of the domain is based on viewing data as a labeled directed graph. We have two kinds of objects in the graphs: logical identifiers, drawn from a set $\mathcal{I}$, and constants (such as integers, strings, URLs), drawn from a set $\mathcal{C}$, which is disjoint from $\mathcal{I}$. The data graph is a set of atomic facts of the form

$$C(o) \quad \text{or} \quad o_1 \rightarrow l \rightarrow o_2,$$

where $o_1 \in \mathcal{I}$, $l \in \mathcal{C}$, $o, o_2 \in \mathcal{I} \cup \mathcal{C}$ and C is a unary relation, called a *collection* name. The fact $C(o)$ denotes that the object o belongs to the unary relation C. The fact $o_1 \rightarrow l \rightarrow o_2$ denotes that the graph contains an arc from o_1 to o_2, and the arc is labeled by l. Note that arcs in the data graph can only emanate from nodes of logical identifiers. One can view the arcs in the graph as representing a binary relation l, and the extension of l contains the tuple (o_1, o_2).

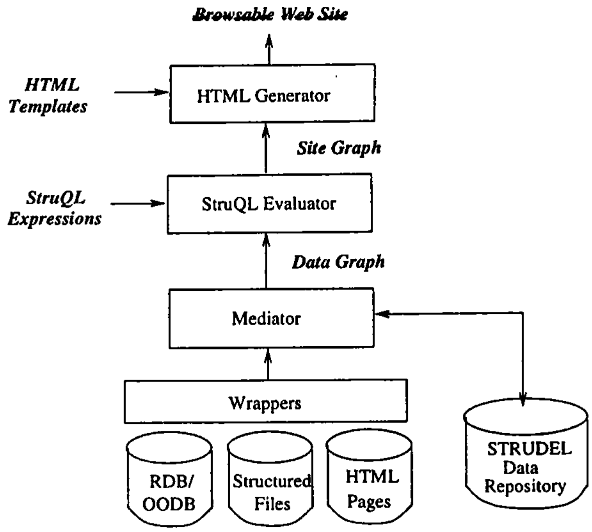


Figure 1: STRUDELArchitecture

The main reason for conceptualizing data in STRUDEL as a directed labeled graph is that STRUDEL ultimately creates Web sites, which are naturally modelled as directed graphs. Note that it is possible to model graphs using a ternary or binary relation, but such a model is not natural when we consider paths in a graph. In addition, a feature of this representation is that the names of the binary relations (i.e., the labels on the arcs) are part of the data, not the schema. As a result, we can accommodate rapidly evolving schema, which is important in this application.

Depending on the Web site being built, the underlying data can be stored in an external source, in STRUDEL's own data repository, or a combination of both. In the former case, STRUDEL requires *wrappers* to access the external sources and to perform the appropriate format translations. Since data may come from multiple sources, STRUDEL requires a data integration component to provide a uniform view of the data. We do not discuss the issue of data integration here, except to mention that STRUDEL uses standard techniques for data integration (see (Arens *et al.* 1996; Levy *et al.* 1996; Ullman 1997; Duschka and Genesereth 1997; Friedman and Weld 1997) for recent works on this topic.)

The STRUQL Language

The STRUQL language is used to describe how a Web site is constructed from the raw data modeled by a data graph. We now describe STRUQL's core. We distinguish two parts of a STRUQL expression: the *query* part and the *construction* part. The query part supports querying of the data graph. The result of applying the query part to the data graph is a relation (i.e., a set of tuples). The construction part uses this relation to construct the nodes and arcs in the output graph. The result of the construction component (and hence of a complete STRUQL expression) is a new graph. We often use expressions that contain only the query part and refer to them as STRUQL-query expressions.

In STRUQL expressions, we distinguish *arc variables* from normal variables. Intuitively, normal variables are bound to nodes in the data graph and arc variables are bound to labels on the arcs We denote arc variables by the capital letter L.

The query part of a STRUQL expression often refers to pairs of nodes in the graph with specific types of paths between them. Such paths are specified by *regular path expressions*. A regular path expression over the set of constants $\mathcal{C}$ is formed by the following grammar (R, R_1 and R_2 denote regular path expressions):

$$R := \epsilon \mid a \mid \text{not } (a) \mid \_ \mid L \mid (R_1.R_2) \mid (R_1 \mid R_2) \mid R^*.$$

In the grammar, a denotes a letter in $\mathcal{C}$; not (a) matches any constant in $\mathcal{C}$ different from a. _ denotes any constant in $\mathcal{C}$, . denotes concatenation, and | denotes alternation. R^*, the Kleene star, can be matched by 0 or more repetitions of R. For example, $a.b.\_.c^*$ denotes the set of strings beginning with ab, then an arbitrary character and then any number of occurrences of c. We use $*$ as a shorthand for $\_^*$, meaning an arbitrary path.

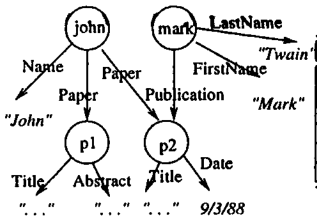
A single-block STRUQL expression has the form:

where $C_1 \wedge \ldots \wedge C_k$,
create $N_1, \ldots, N_n$
link $K_1, \ldots, K_p$
collect $G_1, \ldots, G_q$

All the clauses in a STRUQL expression are optional. The where clause is the query part of the expression, and the other three clauses are the construction part. Each conjunct in the where clause is either of the form $C(X)$ or $X \rightarrow R \rightarrow Y$, where C is a collection name, R is a regular path expression, X is a variable, and Y is a variable or constant in $\mathcal{C}$.

Example 1: Consider the following STRUQL expression:

where $Person(X) \wedge X \rightarrow ('Paper' |' Publication') \rightarrow Y \wedge$
$Y \rightarrow L \rightarrow Z$
create $PersonPage(X), PaperPage(Y)$
link $PersonPage(X) \rightarrow$ *'Paper'* $\rightarrow PaperPage(Y)$,
$PaperPage(Y) \rightarrow L \rightarrow Z$
collect $Page(PersonPage(X)), Page(PaperPage(Y))$



X	Y	L	Z
john	p1	Title	"..."
john	p1	Abstract	"..."
john	p2	Title	"..."
john	p2	Date	9/3/88
mark	p2	Title	"..."
mark	p2	Date	9/3/88

Figure 2: A datagraph and the relation R_Q.

Informally, the where clause considers all quadruplets (X, Y, Z, L), such that X is a person, there exists an arc labeled 'Paper' or 'Publication' from X to Y, and there is an arc from Y to Z. The construction part creates a page for every person X and for every publication Y, adds an arc from the person page to the publication page, and also copies all the arcs emanating from Y to the result graph. Finally, the collect expression adds the new nodes to the *Page* collection.

Semantics: We first explain the semantics of the where clause of a STRUQL expression Q. Consider each substitution ψ from the variables in the where clause to $\mathcal{I} \cup \mathcal{C}$, such that each arc variable is mapped to an element of $\mathcal{C}$, and

- if C_i is of the form $C(X)$, then $C(\psi(X))$ is in the data graph, and
- if C_i is of the form $X \rightarrow R \rightarrow Y$, then there is a path P in the data graph between $\psi(X)$ and $\psi(Y)$ such that P satisfies $\psi(R)$. Here, applying ψ to the regular path expression R replaces all the arc variables in R by constants in $\mathcal{C}$.

Each substitution ψ above defines a tuple whose arity is the number of variables in Q. The set of all such tuples form a relation, which we denote with R_Q, and which is the result of the where clause.

Example 2: Figure 2 illustrates a data graph. The collection *Person* (not shown) consists of the identifiers *john* and *mark* respectively. The result R_Q for the query in Example 1 is also shown.

We now describe the semantics of the construction part of a STRUQL expression. X and Y denote variables in the where clause, and f and g denote function symbols. We only use unary function symbols, however STRUQL supports function symbols of any arity. The create clause specifies the new nodes in the result graph. Each of the N_i's is of the form $f(X)$. For every value a of the X attribute in R_Q, the result graph contain the node $f(a)$.

The link clause specifies the links in the result graph. Each K_i is of the form $f(X) \rightarrow l \rightarrow g(Y)$, where l is

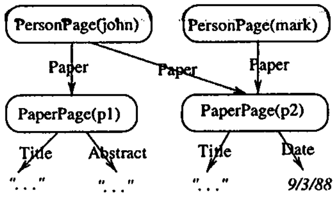


Figure 3: site graph.

a constant in C or an arc variable in the where clause. If l is an arc variable L, then for every triple (a, c, b) in the projection of the attributes (X, L, Y) in R_Q, the result graph contains an arc labeled c from $f(a)$ to $g(b)$. When $l \notin C$, the result is obtained by projecting on the attributes X and Y in R_Q.

Finally, the collect clause specifies the unary facts that hold in the result graph. Each G_i is of the form $D(f(X))$, where D is a collection name (not necessarily from the data graph). The semantics are defined in a similarl fashion, as above. We also implicitly associate a collection in the result graph with every function symbol that appears in the link or create clauses, e.g., if $f(X)$ appears there, then f is also a collection name in the result graph, and every constant in the graph of the form $f(a)$ is in the extension of the collection f.

Example 3: Fig. 3 shows the result of applying the query from Example 1 to the datagraph in Fig. 2.

Above, we described STRUQL expressions with one block. In practice, several blocks are common, and their order does not affect the result graph. We also allow *nesting* of blocks. Nesting makes queries more concise, because a nested where clause inherits all the conditions from the where clauses of its containing blocks. For example, in Figure 5 the where clause on line (12) includes the conditions from line (7). Finally, a block can have multiple create and link clauses, and the result graph is independent of their order.

Example Web Site

To finish our description, we give a simplified example of a researcher's homepage created with STRUDEL. The source of raw data is a Bibtex bibliography that contains the researcher's publications. In the data graph, we represent this data by a collection PUBLICATIONS, as seen in Figure 4. Note that every paper is annotated with one or more categories and with the file names of its abstract and postscript source.

The structure of the homepage site is defined by the STRUQL expression in Figure 5. The site has four types of pages: a root page containing general information, an "All Titles" page containing the list of titles of the researcher's papers, a "category" page containing summaries of papers in a particular category, and a "Paper Presentation" page for each paper.

```
object publ in Publications {
  title "Web Sites With Common Sense"
  author "John McCarthy"
  author "Tim Berners-Lee"
  year   1998
  booktitle "AAAI 98"
  pub-type  "inproceedings"
  abs-file  "abstracts/bm98"
  ps-file   "proceedings/aaai98.ps"
  category  "Philosophical Foundations"
  category  "Knowledge Representation"
}
```

Figure 4: Fragment of data graph for homepage site

The first clause creates the RootPage and AllTitles pages and links them. Lines 7–9 create a page for each publication, and links the publication page to each of its attributes. Note that we copy all the attributes of a given publication using the arc variable L. Lines 12–14 consider the category attribute of each publication and create the appropriate category pages with links to the appropriate publication pages. Finally, lines 18–19 links the "All Titles" page to the titles of all the papers and the papers' individual pages.

Verifying Integrity Constraints

Our goal is to develop algorithms for verifying that a Web site created by STRUDEL satisfies certain constraints. In this section, we formally define the problem. To motivate this goal, consider the following examples of integrity constraints one may wish to enforce on the Web site generated by our example.

1. All *PaperPresentation* pages are reachable from the root page by a path from the root.
2. If a publication's postscript source exists, then its *PresentationPage* is linked to it.
3. Unless you follow the link labeled "Back to Regular Site", no page reachable from "TextOnlyRoot" contains images.[1]

We define the verification problem as an entailment problem of a STRUQL expression and a logical sentence describing the integrity constraint. We express integrity constraints by logical sentences ϕ built from atoms of the form $C(X)$ and $X \rightarrow R \rightarrow Y$, the logical connectives $\wedge, \vee, \neg$, and the quantifiers $\forall$ and $\exists$.

[1]This example is inspired by an inconsistency in the CNN Web site. If you go to the text-only version and click on any article, then you get a page with images, defeating the purpose of the text-only version.

```
INPUT BIBTEX
// Create root page and abstracts page and link them
CREATE RootPage(), AllTitlesPage()
LINK   RootPage() -> "All Titles" -> AllTitlesPage()

// Create a presentation for every publication x
WHERE  Publications(X), X -> L -> V
CREATE PaperPresentation(X)
LINK   PaperPresentation(X) -> L -> V,

// Create a page  for every category
  { WHERE  L = "category"
    CREATE CategoryPage(V)
    LINK   CategoryPage(V)  -> "Paper" -> PaperPresentation(X), CategoryPage(V)  -> "Name" ->  V

    // Link root page to each category page
           RootPage()  -> "CategoryPage"  -> CategoryPage(V) }
  { WHERE L = "title"
    LINK  AllTitlesPage() -> "title" -> V, AllTitlesPage -> "More Details" -> PaperPresentation(X) }
OUTPUT HomePage
```

Figure 5: Site definition query for example homepage site

Given a labeled, directed graph G, we can determine whether G satisfies a sentence ϕ by interpreting G as a logical model. That is, if A is an atom, and $A \notin G$, then $\neg A$ holds in the model. In addition, the only constants in the domain are those that appear in G, hence, we can evaluate a universally quantified formula.

Given a data graph G, let $Q(G)$ denote the site graph that results from applying the STRUQL expression Q to G. Now we can define the verification problem.

Definition 1: *We say that the integrity constraint ϕ is satisfied by Q if for any given data graph G, the sentence ϕ is satisfied in the graph $Q(G)$.*

Note that the definition requires that ϕ be satisfied in *all* possible sites created by Q and is not specific to a particular data graph.

Example 4: The following sentences represent the three examples above.

1. $(\forall X) PaperPresentation(X) \Rightarrow RootPage() \rightarrow * \rightarrow X$
2. $(\forall X.\forall Y)(Publication(X) \wedge X \rightarrow \text{"psFile"} \rightarrow Y) \Rightarrow PaperPresentation(X) \rightarrow * \rightarrow Y$.
3. $(\forall X, Y) TextOnlyRoot(X) \wedge$ $X \rightarrow (\text{not}\ (\text{"BackToRegularSite"}))^{*}.\text{"Image"} \rightarrow Y$ $\Rightarrow false$.

Verification Algorithm

The previous section gave a very general formalization of the problem of verifying integrity constraints. In this section, we present an algorithm for verifying integrity constraints that captures a large class of constraints that occur in practice. A closer study of these integrity constraints shows that the sentence ϕ often has the more specific form $Q_1 \Rightarrow Q_2$, where Q_1 and Q_2 are conjunctive formulas. For instance, in the first example, Q_1 is the formula $PaperPresentation(X)$ and Q_2 is $RootPage() \rightarrow * \rightarrow X$.

One main problem in developing an algorithm for reasoning about constraint formulae is that they often refer to the site graph, instead of the data graph. Recall that the site graph is defined by a STRUQL expression Q over the data graph. In 1 and 3 of Example 4, Q_1, Q_2 refer to the site graph; in (2), Q_1 refers to the data graph).[2] In the former cases, we need to consider the composed formulae $Q_1 \circ Q$ and $Q_2 \circ Q$ which are on the data graph. The key idea of our algorithm is to translate these composed formulae into simpler ones. As a result, we can reduce the verification problem to a reasoning problem on certain types of Horn theories, for which sound and complete reasoning algorithms are known.

To perform the translation, we use a novel data structure, the *site schema*, that provides a schematic graphical representation of a STRUQL expression. Due to space limitations, we consider only a simplified form of site schema. The site schema for the homepage Web site is shown[3] in Figure 6. The site schema G_Q for a STRUQL expression Q is a labeled directed graph, that describes the *possible* paths in a Web site resulting from the expression Q. The graph G_Q contains a node N_f for every function symbol f appearing in Q,

[2]Syntactically, we cannot distinguish between expressions referring to the site graph or the data graph, unless the expression mentions function symbols or collections defined in the STRUQL expression. In other cases, we assume that the expression refers only to the data graph.

[3]To avoid clutter we removed two edges and replaced some conditions with simpler, equivalent ones.

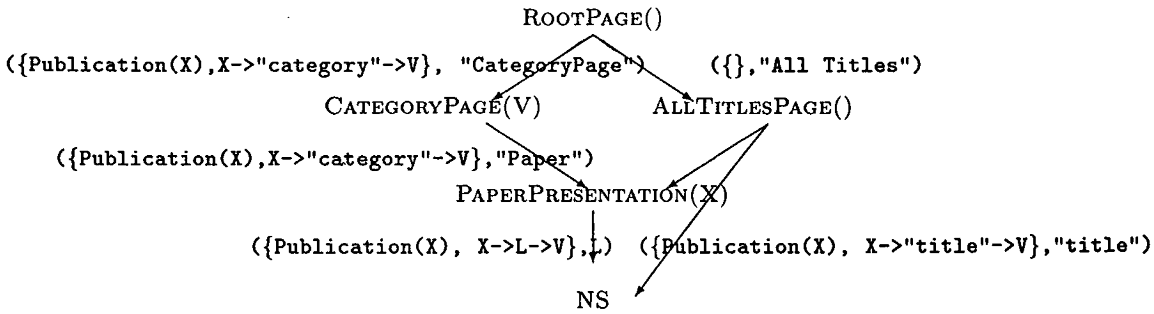


Figure 6: The site schema of the homepage site.

which corresponds to nodes of the form $f(a)$ in the site graph, and a special node, NS, which corresponds to non-Skolem nodes in the site graph.

The graph's links are annotated with conditions (i.e., where clauses) that guarantee the existence of a link between nodes. Specifically, given a link clause K, let K_W denote the where clause that applies to K; recall that if K is nested, then K_W includes all the conditions of the containing where clauses. For every atom in K of the form $f(X) \to l \to g(Y)$, we add an arc from N_f to N_g labeled (K_W, l). Multiple arcs with different labels may exist between N_f and N_g. If the link is of the form $f(X) \to l \to v$, where v is a variable, then we add an arc from N_f to NS labeled (K_W, v).

Given the site schema, the next step of the algorithm is to describe conditions for the existence of more complex paths by juxtaposing conditions on single edges. The important point is that the conditions for the complex paths refer only to the data graph, not the site graph. For example, for any pair of nodes N_f and N_g in the site schema, we can write a formula describing the conditions for the existence of an arbitrary path from N_f to N_g or for the existence of a path from N_f to N_g of length at most n.

Example 5: In our example, the following formula describes the condition for existence of a path from ROOTPAGE() to PAPERPRESENTATION(X):

$(Publication(X) \wedge X \to \text{"category"} \to v) \vee$
$(Publication(X) \wedge X \to \text{"title"} \to v)$

The first disjunct describes the path that may go through CATEGORYPAGE(V), and the second describes the path going through ALLTITLESPAGE(). Note that we removed some redundant conditions in the formula. Hence, to verify that every publication page is reachable from the root page, we need to check the validity of the following sentence:

$$Publication(X) \Rightarrow [(Publication(X) \wedge X \to \text{"category"} \to v) \vee (Publication(X) \wedge X \to \text{"title"} \to v)].$$

Suppose we want to write a condition that expresses the existence of a path from ROOTPAGE() to PAPERPRESENTATION(X) that *does not* go through ALLTITLESPAGE. In this case, we only consider paths in the site schema that do not go through ALLTITLESPAGE, and hence the condition is simply

$(Publication(X) \wedge X \to \text{"category"} \to v)$.

More generally, whenever Q is a *StruQL* expression with a cycle-free site schema and Q_1 is a conjunctive formula on the site graph, we can compute a new formula equivalent to $Q_1 \circ Q$, which is a disjunction of conjunctive formulae (i.e., a set of nonrecursive Horn rules). Similarly, one can show that, if Q is an arbitrary *StruQL*-query expression (not necessarily cycle-free) and Q_1 a conjunctive formula that does not contain the Kleene star, then $Q_1 \circ Q$ is equivalent to a disjunction of conjunctive formulae. These technique allow us to express the composed formulae $Q_1 \circ Q$ and $Q_2 \circ Q$ as disjunctions of conjunctive formulae.

We can now present the main results. In the following theorems, Q is a *StruQL*-expression defining a site graph from a data graph, and Q_1, Q_2 are conjunctive formulae defining the constraint $Q_1 \Rightarrow Q_2$ on the site graph. The theorems distinguish between the cases in which the site schema does and does not contain cycles. As mentioned before, Q_1, Q_2 can be expressed either on the data graph or on the site graph. Finally, the computational complexity of the verification algorithms are w.r.t. the size of Q, Q_1, and Q_2, and *not* the size of the data or site graphs.

Theorem 1: *Let G_Q be the site schema of the* STRUQL *expression Q, and assume that G_Q is acyclic. Then, the problem of verifying the constraint $Q_1 \Rightarrow Q_2$ is decidable, and the complexity of the decision problem is in exponential space. Moreover, if all regular expressions in Q, Q_1, Q_2 are* simple, *i.e., they are restricted to the form $R_1.R_2 \ldots R_n$, where each R_i is either a label or $*$, then the decision problem is in NP.*

Theorem 2: *Assume that either Q_1 is expressed only on the data graph, or that Q_1 does not contain the Kleene star. Then, the problem of verifying the constraint $Q_1 \Rightarrow Q_2$ is decidable, and the complexity of the decision problem is in NP w.r.t. the size of Q_1.*

It is important to note that Theorems 1 and 2 combined capture many cases encountered in practice for which the resulting algorithm can be implemented relatively efficiently.

The proof of Theorem 1 proceeds by reducing the verification problem to a logical entailment problem for STRUQL-query expressions, which is known to be decidable (Florescu *et al.* 1998); the case for simple regular expressions has been shown to be in NP. The proof of Theorem 2 proceeds by a reduction to the problem of entailing a datalog expression from a non-recursive datalog expression, which has been shown to be decidable in (Cosmadakis and Kanellakis 1986).

Conclusions and Related Work

We considered the problem of expressing integrity constraints on the structure of Web sites and verifying whether they hold given a declarative specification of the site. We have only considered the problem of verifying whether or not a constraint holds. A subsequent question is how to *fix* a STRUQL specification when a constraint does not hold. One important benefit of our algorithm is that it returns a *counter-example* data graph when the constraints are not satisfied. Thus, the site builder can decide whether the constraint was not specified well or whether the STRUQL specification needs to be changed. For instance, in Example 5, if a publication does not have a category or title, it will not be reachable from the root page. The site builder may decide that this is acceptable or that the system must enforce that every publication has a category.

Our work is most related to the problem of verifying rule-based knowledge base systems. (Levy and Rousset 1996) show how to reduce the verification problem to one of entailment on Horn-rule formulas. STRUQL is a different formalism from the one used in that paper, therefore the challenge was to find the cases, revealed by the site schema, in which there is a a similar reduction. (Schmolze and Snyder 1997) considers a similar problem, but with rules that may have side effects. Such rules do not exist in our formalism. (Rousset 1997) proposes an *extensional* approach to verifying constraints on Web sites. Constraints are expressed in a rule-based language, but they are checked against the current state of the Web-site at any given moment, similar to the way integrity constraints would be checked when a database is updated.

The site schema is an elaboration of *graph schemas*, introduced in (Buneman *et al.* 1997) for query optimization. Site schemas contain more information than graph schemas and are derived automatically from the STRUQL expression. In addition, we show how to use the structure for integrity-constraint verification. Similar data structures have been used for describing interactions among Horn rules (e.g., (Etzioni 1993; Levy *et al.* 1997)), but none of them have been used for verification.

The main issue for future research is finding larger classes of constraints for which verification is possible. At the time of writing, the question of decidability of entailment between two STRUQL-query expressions over the site graph is still open. Answering that question will lead to a larger class of verifiable constraints.

References

Arens, Yigal; Knoblock, Craig A.; and Shen, Wei-Min 1996. Query reformulation for dynamic information integration. *International Journal on Intelligent and Cooperative Information Systems* (6) 2/3:99–130.

Buneman, Peter; Davidson, Susan; Fernandez, Mary; and Suciu, Dan 1997. Adding structure to unstructured data. In *ICDT*, Deplhi, Greece. Springer Verlag. 336–350.

Cosmadakis, S. and Kanellakis, P. 1986. Parallel evaluation of recursive rule programs. In *ACM PODS.*

Duschka, Oliver M. and Genesereth, Michael R. 1997. Query planning in infomaster. In *Proceedings of the ACM Symposium on Applied Computing, San Jose, CA.*

Etzioni, Oren 1993. Acquiring search-control knowledge via static analysis. *Artificial Intelligence* 62.

Fernandez, Mary; Florescu, Daniela; Kang, Jaewoo; Levy, Alon; and Suciu, Dan 1998. Catching the boat with strudel: experience with a web-site management system. In *Proceedings of SIGMOD.*

Florescu, Daniela; Levy, Alon; and Suciu, Dan 1998. Query containment for conjunctive queries with regular expressions. In *Proceedings of the Symposium on Principles of Database Systems, PODS–98.*

Friedman, M. and Weld, D. 1997. Efficient execution of information gathering plans. In *Proceedings of IJCAI.*

Guha, R.V. 1997. Hotsauce MCF. http://mcf.research.apple.com/hs.

Levy, Alon Y. and Rousset, Marie-Christine 1996. Verification of knowledge bases using containment checking. In *In Proceedings of AAAI.*

Levy, Alon Y.; Rajaraman, Anand; and Ordille, Joann J. 1996. Query answering algorithms for information agents. In *In Proceedings of AAAI.*

Levy, Alon Y.; Fikes, Richard E.; and Sagiv, Shuky 1997. Speeding up inferences using relevance reasoning: A formalism and algorithms. *Artificial Intelligence* 97(1-2).

Perkowitz, Mike and Etzioni, Oren 1997. Adaptive web sites: an AI challenge. In *In Proceedings of IJCAI.*

Rousset, Marie-Christine 1997. Verifying the web: a position statement. In *Proceedings of the 4th European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV-97).*

Schmolze, James and Snyder, Wayne 1997. Detecting redundant production rules. In *In Proceedings of AAAI.*

Ullman, Jeffrey D. 1997. Information integration using logical views. In *Proceedings of the International Conference on Database Theory.*