VLDB manuscript No.
(will be inserted by the editor)

Declarative Specification of Web Sites with STRUDEL

Mary Fernández[1], **Daniela Florescu**[2], **Alon Levy**[3], **Dan Suciu**[4]

[1] AT&T Labs Research, e-mail: `mff@research.att.com`
[2] INRIA Roquencourt, e-mail: `dana@rodin.inria.fr`
[3] Univ. of Washington, e-mail: `alon@cs.washington.edu`
[4] AT&T Labs Research, e-mail: `suciu@research.att.com`

Received: date / Revised version: date

Abstract STRUDEL is a system for implementing *data-intensive* Web sites, which typically integrate information from multiple data sources and have complex structure. STRUDEL's key idea is separating the management of a Web site's data, the specification of its content and structure, and the visual representation of its pages. STRUDEL provides a declarative *query language* for specifying a site's content and structure, and a simple *template language* for specifying a site's HTML representation. This paper contains a comprehensive description of the STRUDEL system and details the benefits of declarative site specification. We describe our experiences using STRUDEL in a production application and describe three different, but complementary, systems that extend and improve upon STRUDEL's original ideas.

Key words Web-site management, declarative query languages

1 Introduction

Web sites have become the principal mechanism for disseminating and accessing information on the Internet and on corporations' high-speed intranets. Before intranets, access to geographically dispersed information systems was usually limited to those people who administered the systems locally. In this environment, *data integration*, the task of integrating information from multiple data sources, was difficult, if not impossible. Because of their value to diverse groups in a company, integrated information systems must be easily accessible and therefore, are usually realized as Web sites.

Correspondence to: Mary Fernández, AT&T Labs - Research, 180 Park Ave., Room E243, Florham Park, NJ 07932.

These *data-intensive* Web sites usually integrate information from multiple data sources, often have complex structure, and present increasingly detailed views of data, from a summary perspective at a top-level page to a detailed perspective at a lower-level page.

As the demand for data-intensive Web sites increases, the demand for tools to help create and maintain such sites also increases. Because data-intensive Web sites are typically hard to specify and implement, commercial vendors and academic researchers are actively developing methods and tools for building sites. Although the goals of the myriad solutions differ, most attempt to isolate and automate the common tasks of Web-site development. These include: (1) choosing and accessing the data that will be displayed at the site; (2) designing the site's content, i.e., specifying the data contained within each page; (3) designing the site's structure, i.e., specifying the navigational links between pages; and (4) designing the visual presentation of pages.

In common practice, data-intensive Web sites usually are not implemented by automated tools, but by groups of loosely related programs written in imperative scripting languages, such as Perl. Scripting languages are well-suited for "gluing" together other software components [24], which makes them popular for constructing Web sites. The scripts for many site implementations, however, interleave the code for data access and integration, page construction, and HTML generation. As a result, important site-management tasks, such as automatically updating or restructuring a site, optimizing a site's performance based on common page-access patterns, or enforcing integrity constraints on a site's structure, are tedious to perform and difficult to automate.

In this work, we argue that implementing data-intensive Web sites is primarily a data-management problem, whose solution consists of three main programming tasks: *accessing and integrating* the data available in the site, *building* the site's content and structure, i.e., specifying the data in each page and the links between pages, and *generating* the HTML representation of pages. To better support these tasks, we developed the STRUDEL system [11]. STRUDEL's key idea is separating the management of a Web site's data, the specification of its content and structure, and the HTML representation of its pages. STRUDEL provides a declarative *query language*, called STRUQL, for specifying the content and the structure of a Web site, and a simple *template language* for specifying a site's HTML representation. STRUDEL's query interpretor automatically derives the site from a STRUQL query. STRUDEL has many benefits: explicit separation of the three programming tasks allows multiple versions of a site to be derived from one specification [11], and STRUQL's semantics supports verification of integrity constraints on a site's structure [13].

The contribution of this work is a comprehensive description of the STRUDEL system and a discussion of how STRUDEL's most important ideas, separation of tasks and declarative specification, have made subsequent research possible. The paper coalesces content previously presented in three separate publications [11–13]. We focus on STRUDEL's architecture for separating data access, site specification, and page generation, and describe its simple, semistructured data model, which can support both structured and unstructured data sources. We describe in detail StruQL, STRUDEL's declarative query language and show that it is as expressive as first-order logic plus transitive closure. To illustrate the value of declarative specification, we present an algorithm for verifying structural constraints on a Web site given its declarative definition in STRUQL.

During the past two years, our experience with STRUDEL has initiated three distinct, but complementary, areas of research. The STRUDEL-R system [18] investigates strategies for optimizing the run-time generation of a site given a declarative specification of the site over relational data sources. The Fun-STRUDEL system [14] focuses on the software-engineering problem of producing site implementations that are extensible, reusable, analyzable, and optimizable; it extends STRUQL with query functions to improve the modularity and re-usability of site definitions and with declarative forms to support dynamically bound inputs. The TIRAMISU system [3] provides a declarative site-specification language that is decoupled from specific implementation tools; any implementation tools that support Tiramisu's common application-program interface can be used together to implement a site. We describe these new systems and discuss how they extend and improve upon STRUDEL's original ideas.

We compare STRUDEL to several classes of site-creation systems, including site-design tools based on hypermedia design methodologies [6,25,27], server-side scripting languages, and site-implementation tools that are applications of database research [4,8]. We also discuss how STRUDEL relates to the use of XML-based technologies for implementing Web sites.

We emphasize that STRUDEL is a site-implementation tool, not an environment for Web-site design, nor is it intended for non-technical users or for development of any Web-based application. Although the implementations of STRUDEL and its successors are prototypes, not commercial products, a production Web site in AT&T was implemented using Fun-STRUDEL. We present a brief case study in which we compare the site's original implementation with its complete re-implementation in Fun-STRUDEL and show that the new implementation is much smaller, more reusable, and unlike the original site, can be analyzed and optimized. Based on our experience, we have found that STRUDEL is best suited for data-intensive, non-transactional Web sites.

The paper is organized as follows. Section 2 describes STRUDEL's architecture and data model. Section 3 introduces STRUQL's syntax through several examples, describes its semantics, and gives a formal proof of its expressive power. STRUDEL's template language is described in Section 4. Section 5 presents an algorithm for verifying integrity constraints on a site, which relies on STRUQL's declarative semantics. We compare STRUDEL to related systems in Section 6. In Sections 7 and 8, we describe STRUDEL's descendents and present the case study.

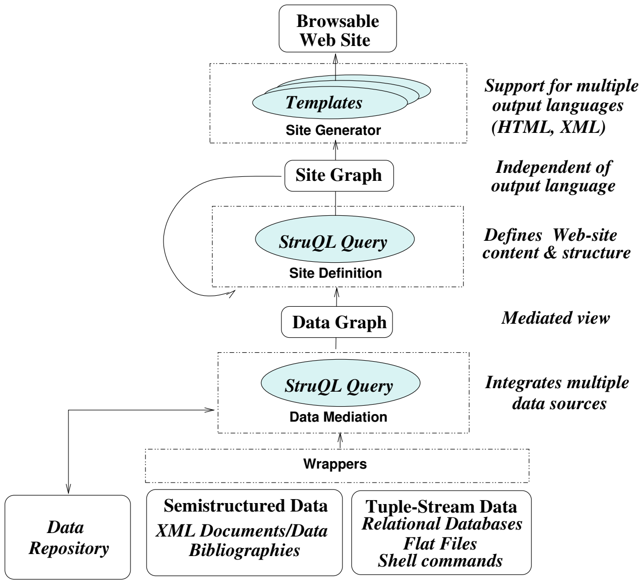

Fig. 1 STRUDEL Architecture

2 STRUDEL **Architecture**

We first describe STRUDEL's architecture, depicted in Figure 1, before focusing on its query language. In Fig. 1, rectangles depict processes and emboldened terms specify the inputs and outputs of the processes.

2.1 Data model and exchange format.

The foundation of STRUDEL is the semistructured data model. Semistructured data is characterized as having few type constraints, a rapidly evolving schema, or missing schema [1] and is typically modeled by a labeled, directed graph. This data model is appropriate for STRUDEL, because Web sites are graphs with irregular structure and non-traditional schemas. Furthermore, semistructured data facilitates integration of data from multiple, non-traditional sources. In every level of the STRUDEL architecture, data is modeled as a labeled, directed graph. STRUDEL's data model is a variation of the OEM data model.

A STRUDEL graph is a set of nodes or *objects*, in which each object is either *complex* or *atomic*. A complex object is a set of *(attribute, object)* pairs, and an atomic object is an atomic value (e.g., `int`, `string`, `mpeg`). Hence, edges in a data graph are labeled by attributes and leaves labeled with atomic values. STRUDEL supports the atomic types integer, float, string, date, and mime-content types such as URL, image, html, and postscript. Internal nodes have unique object identifiers, called *OIDs*. Objects are grouped into named *collections*, which are referenced in queries.

Objects may belong to multiple collections, and objects in the same collection may have different representations. We use the terms nodes and objects interchangeably, but note that object does not denote a strictly typed value as it does in an object-oriented language or database.

Graphs are stored in STRUDEL's data repository (bottom-left of Fig. 1). The repository's initial data may be obtained from wrappers that convert data in external sources into the graph format. STRUDEL provides wrappers for data sources commonly accessed by Web sites, e.g., relational databases that support JDBC, BibTeX bibliographies, flat text files, and XML documents (bottom of Fig. 1).

Data is exchanged between the data repository and external sources in XML [29]. Initially, STRUDEL used a home-grown syntax for data exchange, but when XML became available, we migrated to that syntax. Figure 2 contains a fragment of a bibliography in XML. XML element tags correspond to attributes, i.e., graph edges; an element's complex or atomic value corresponds to an object, i.e., a graph node. We use graph terms, e.g., attributes and objects, when describing example data, instead of XML terms, e.g., elements. The complex objects `article` and `inproceedings` have object identifiers, specified by their `id` values: `pub1` and `pub2`, respectively. In semistructured data, the names, types, and cardinality of attributes may vary. For example, the `article` element has two `category` attributes, but the `inproceedings` element has only one; the `article` has a `journal` attribute, but the `inproceedings` element has a `booktitle` attribute. By default, the root node of an XML data source is contained in the graph collection named `XMLRoot`.

2.2 Data mediator.

STRUDEL's mediator supports data integration by providing a uniform view of all underlying data, irrespective of where it is stored. The mediated view, called a *data graph* (center of Fig. 1), is specified as a STRUQL query over the data sources. When designing the mediator, we addressed two problems: whether to warehouse data from external sources or to access the external sources on demand at query time (see [22] for a comparison); and how to specify the relationship between the attributes and collections in the mediated schema and those in the data sources (see [28] for a discussion of possible approaches). In STRUDEL, we implemented warehousing; the result of data integration is stored in STRUDEL's data repository. This simplified our implementation and sufficed for applications that have small data sources. STRUDEL's architecture, however, can accommodate either approach. In practice, a mediated schema and intermediate data graph are often unnecessary, because the site graph can be defined by a query that references the data sources directly; that is, the data graph is just the disjoint union of all the graphs returned by the wrappers. This is the case when the site builder chooses to integrate the data sources and define the site in one step.

```
<bibliography>
 <article id="pub1">
   <title>Specifying Representations...</title>
   <author><firstname>Norman</firstname>
           <lastname>Ramsey</lastname></author>
   <author><firstname>Mary</firstname>
           <lastname>Fernandez</lastname</author>
   <year>1997</year>
   <month>May</month>
   <journal>Transactions on Programming...</journal>
   <abstract>abstracts/toplas97.txt</abstract>
   <postscript>papers/toplas97.ps.gz</postscript>
   <volume>19</volume>
   <number>3</number>
   <category>Architecture Specifications</category>
   <category>Programming Languages</category>
 </article>

 <inproceedings id="pub2">
   <title>Optimizing Regular...</title>
   <author><firstname>Mary</firstname>
           <lastname>Fernandez</lastname</author>
   <author><firstname>Dan</firstname>
           <lastname>Suciu</lastname></author>
   <year>1998</year>
   <booktitle>Proc. of ICDE</booktitle>
   <abstract> abstracts/icde98.txt</abstract>
   <postscript>papers/icde98.ps.gz</postscript>
   <category>Semistructured Data</category>
 </inproceedings>
</bibliography>
```

Fig. 2 Fragment of BibTex data for example site

2.3 Query processor.

Strudel provides a declarative language, StruQL (Site TRansformation Und Query Language) for querying and restructuring semistructured data. A StruQL query (two middle boxes in Fig. 1) is applied to a data graph and its result is a *site graph*, which represents the site's content and structure and is completely independent of the output language. StruQL queries are compositional: a site graph is another example of semistructured data and can be queried by another StruQL query. As in traditional query processing, a query is first translated by the query optimizer into an efficient physical-operation tree. Strudel's query interpreter includes conventional physical operators as well as those necessary to query the schema (e.g., scan all the attribute names in a graph). In Strudel's first implementation, we built a simple heuristic-based optimizer, which tries to put operations with small results (e.g., a scan of a small collection, a predicate that reduces the result size) early in the plan and delays expensive operations (e.g., joins) until late in the plan. The optimizer only transforms the StruQL query on the data graph, not the queries that access the external sources. Later, we developed a more comprehensive cost-based optimization algorithm [15]. The newer optimizer can enumerate plans that exploit indexes on the data and the schema in order to choose the best plan. The optimizer is also well suited for accessing data in external sources when only limited access patterns are supported, such as form-based interfaces to structured data sources, because they often require that some inputs be given to access the data, and therefore pose novel challenges to query optimization.

2.4 Site generator.

To produce a browsable Web site, an *HTML template* is associated with each object in the site graph. Objects in a site graph may represent complete pages or page components. Usually, a template is associated with a collection of related objects. A template interleaves plain HTML text with STRUDEL-specific tagged expressions that access an object's attributes and format attributes' values. The template language is similar to other languages that separate presentation from content [5,29,7]. This technique simplifies the site programmer's task: he writes plain HTML extended with simple programmatic constructs, instead of a more complex scripting program that generates HTML. STRUDEL's generator (top of Fig. 1) evaluates the appropriate template for every object in a site graph. The resulting pages are the browsable Web site. Section 4 describes the template language in detail.

3 STRUQL Query Language

In STRUDEL, we need to query graphs and create new graphs at the *mediation* level, when data from different external sources is integrated into a data graph, and at the *site-definition* level, when site graphs are constructed from a data graph. We use STRUQL at both levels. Figure 3 contains the EBNF grammar for STRUQL. Our aim is to describe STRUQL's semantics by example, not to advocate a particular syntax; alternatives are discussed in Section 6.3.

A STRUQL query defines a *function* from one or more input graphs to an output graph. Each STRUQL query has two parts: a query part and a construction part. The *query part* consists of the where clause and produces all bindings of node and edge variables to values in the data graph that satisfy all predicates in the where clause. The result of "evaluating" the where clause on a given graph is best understood as a relation R, with one attribute for each variable; we define R formally later. The *construction* part consists of the link and collect clauses. It creates nodes, edges, and

$$
\begin{array}{lll}
Query & :- & Block \\
Block & :- & \{ \left[\textsf{where} \left\{ Predicate, \right\} Predicate \right] \\
 & & \left[\textsf{link} \left\{ LinkExpr, \right\} LinkExpr \right] \\
 & & \left[\textsf{collect} \left\{ CollectExpr, \right\} CollectExpr \right] \\
 & & \left\{ Block \right\} \\
 & & \} \\
LinkExpr & :- & SkolemTerm \rightarrow (Var \big| Constant) \rightarrow Term \\
CollectExpr & :- & CollectionName\{(SkolemTerm \big| Var)\} \\
Predicate & :- & Var \rightarrow (Var \big| PathExpr) \rightarrow Term \\
 & \big| & BoolPred \\
Term & :- & Var | Literal \\
SkolemTerm & :- & ID(\left[\left\{ Term, \right\} Term \right]) \\
PathExpr & :- & UnaryPred \\
 & \big| & (PathExpr.PathExpr) \\
 & \big| & (PathExpr \big| PathExpr) \\
 & \big| & PathExpr* \\
BoolPred & :- & CollectionName\{Var\} \\
 & \big| & AtomicPredicate \ \textsf{//e.g.,}\ x < 5, \textsf{l in} \ \{ \textsf{"black"}, \textsf{"white"} \} \\
 & \big| & (BoolPred \ \textsf{and} \ BoolPred) \\
 & \big| & (BoolPred \ \textsf{or} \ BoolPred) \\
 & \big| & \textsf{not} \ (BoolPred)
\end{array}
$$

Fig. 3 EBNF Grammar for StruQL

collections in the result graph for each tuple in the relation R. The result of the complete StruQL query is a new output graph.

For example, the following query returns all PostScript papers directly accessible from home pages:

```
where    HomePages{p}, p ->"Paper"-> q, typeOf(q,"postscript")
collect  PostscriptPages{q}
```

`HomePages` is a collection, `"Paper"` is an edge label, and `typeOf(q, "postscript")` tests whether node `q` is a PostScript file. The condition `p -> "Paper" -> q` means that there exists an edge labeled `"Paper"` from `p` to `q`. The query constructs a new collection, `PostscriptPages`, consisting of all answers.

Each predicate in a where clause is either :

- a path expression, e.g., `p -> "Paper" -> q`, or
- a collection-membership predicate, e.g., `HomePages{p}`, or

– a built-in or external predicate applied to nodes or edges, e.g., `typeOf(p, "postscript")`, or
– a boolean combination of the above.

A path expression has the general form `x -> Var -> y`, which means there exists a single edge from node `x` to node `y` whose value is bound to the variable `Var`, or `x -> R -> y`, which means there exists a path from node `x` to node `y` that matches the regular-path expression `R`. For efficiency, path expressions cannot be combined with boolean connectives. For example, we cannot write:

```
where Bib(x), x -> "Paper" -> y, not(y -> "Title" -> z)
```

Sec. 3.2 explains how composed queries avoid these limitations. Atomic conditions like collection membership and built-in predicates can be combined with the boolean operators `and`, `or`, `not`.

Path expressions include regular expressions, but are more general, because they permit unary predicates on edges. For example, `isName*` is the regular-path expression denoting any sequence of labels such that each satisfies the `isName` predicate, i.e., it matches a path labeled $\mathtt{l}_1.\mathtt{l}_2\ldots\mathtt{l}_n$ provided that all expressions $\mathtt{isName}(\mathtt{l}_1), \mathtt{isName}(\mathtt{l}_2), \ldots, \mathtt{isName}(\mathtt{l}_n)$ are true. In particular, *true* denotes any edge label, and *true*`*` any path; we abbreviate the latter with `*`. Each string constant is considered a predicate satisfied only by that string. Thus, the path expression `"paper"."title"` matches precisely the path `"paper"."title"`. Predicates can be combined with boolean connectors, e.g. `"paper".(not("image")*)` matches any path $\mathtt{l}_1.\mathtt{l}_2\ldots\mathtt{l}_n$ s.t. $\mathtt{l}_1 =$ `"paper"` and $\mathtt{l}_2 \neq$ `"image"`, $\ldots$, $\mathtt{l}_n \neq$ `"image"`. Regular-path expression also include path concatenation and alternation operators (see Fig. 3).

The `link` clause creates a new graph from existing graphs. The new nodes are Skolem terms, and the `link` clause specifies the new edges between these nodes. For example, the following query constructs a new node `HomePage(a)` for every binding of `a` to an author, and links it to the author's name and publications:

```
where Root{r}, r -> "pub" -> p, p -> "author" -> a
link  HomePage(a) -> { "name" -> a, "paper" -> p }
```

Because creating multiple edges from a single node is a common idiom, the `link` expression above is an abbreviation for :

```
link  HomePage(a) -> "name" -> a,
      HomePage(a) -> "paper" -> p
```

`HomePage` is a *Skolem function*. Its semantics is that it creates a new node for every value of `a`. For example, if `a` is bound to `"Jones"` and `"Smith"`, two new nodes named `HomePage("Jones")` and `HomePage("Smith")` are created. For each binding of `a`, two new edges, labeled `"name"` and `"paper"`, are added. If `a` is bound to the same value more than once, no new nodes are created; instead, two new edges are added to the unique node `HomePage(a)`.

For example, "Smith" could be a coauthor of two papers, with OID's p348 and p838. The first time a is bound to "Smith" the following two edges are created:

```
HomePage("Smith") -> { "name" -> "Smith", "paper" -> p348 }
```

The next time a is bound to "Smith", the following edges are created:

```
HomePage("Smith") -> { "name" -> "Smith", "paper" -> p838 }
```

The combined effect is that HomePage("Smith") has three edges:

```
HomePage("Smith") -> { "name" -> "Smith", "paper" -> p348,
                       "paper" ->  p838 }
```

Note that the first edge occurs only once, because value edges are *(node, label, value)* triplets, and duplicates are eliminated.

Label variables. Label variables are bound to labels, not OIDs or strings. For example, the following query creates a page for each paper, and includes all the paper's attributes:

```
where Root{r}, r -> "pub" -> p, p -> L -> v
link  PaperPage(p) -> L -> v
```

Sometimes we want to control which attributes are copied to the new graph. For example, in:

```
where Root{r}, r -> "pub" -> p, p -> "title" -> t,
      p -> L -> v, L in {"author", "year", "journaltitle"}
link  PaperPage(p) -> { "title" -> t,  L -> v }
```

only "title", "author", "year", "journaltitle" are copied to the output graph. Note that "title" is mandatory, but only one of the other three is required. This query is different from:

```
where Root{r}, r -> "pub" -> p, p -> "title" -> t,
               p -> "author" -> a, p -> "year" -> y,
               p -> "journaltitle" -> jt
link  PaperPage(p) -> { "title" -> t, "author" -> a,
                        "year" -> y,  "journaltitle" -> jt }
```

which creates pages *only* for papers that have all four attributes.

For efficiency, label variables cannot be used in conjunction with regular-path expressions. For example the following condition is not allowed:

```
where  x -> "a".("b".L)*."c" -> y    /* ERROR */
```

Sec. 3.2 shows how composed StruQL queries can overcome this limitation. To evaluate this condition, all possible bindings of L to labels must be tried. For example, when L is bound "d", then we must compute the regular-path expression x -> "a".("b"."d")*."c" -> y. It is possible to restrict the number of bindings for L; for example, we could compute the path expression x ->"a"."b" -> z first, then bind L only to labels leaving the node z, but this complicates the query processor. In addition, we have a problem defining the query's semantics. Assume that there is a link statement:

```
where  x -> "a".("b".L)*."c" -> y    /* ERROR */
link   x -> L -> y
```

and assume there are nodes `x, y` connected by a path `x -> "a"."c"-> y`. This leaves L unbound, and it is not clear how to execute the link clause. Under the active domain semantics, L should be bound to all labels in the input graph, which results in an inefficient computation. For these reasons, we decided not to allow variables in conjunction with regular-path expressions. Nevertheless, such expressions can be computed with composed STRUQL queries (see Sec. 3.2).

Skolem Functions. Skolem functions can have an arbitrary number of arguments. A nullary Skolem term defines a unique node. The following example constructs a home page for each author, then groups the author's publications by year:

```
where Root{r}, r -> "pub" -> p,
      p -> "author" -> a, r -> "year" -> y
link  NewRoot() -> "person" -> HomePage(a)
      HomePage(a) -> {"name" -> a,
                      "entry" -> YearEntry(a, y)},
      YearEntry(a, y) -> "paper" -> p
```

The `link` clause may contain links between both new and old nodes: for example `YearEntry(a,y)` is a new node, but `p` is an existing node in the data graph. The only constraint is that new links cannot be added to old nodes. For example, the following is prohibited:

```
link  p -> "authorHomePage" -> HomePage(a)  /* ERROR */
```

This is consistent with the functional semantics of languages like Lisp and ML, in which input values are immutable. A STRUQL query is not well-defined nor guaranteed to terminate on a data graph that is mutable, but is guaranteed to terminate on immutable input graphs. This restriction also prohibits where clauses from ranging over collections defined in other blocks, because collections are defined with respect to the output graph, not the input graphs. Thus, the output graph always points to the input graph; this is just a convenience, since the input graph can always be copied into the output graph, as the following example illustrates.

This query produces a site graph that excludes any nodes that contain image files:

```
where    Root{p}, p -> * -> q,
         q -> l ->  q', not(typeOf(q', "image"))
link     NewNode(q) -> l -> NewNode(q')
collect  TextOnlyRoot{NewNode(p)}
```

Here we use a regular-path expression, `*`, to reach all nodes `q` accessible from `p`. In effect, the query above copies the entire graph except nodes with image type. Note that we can copy the graph without knowing its structure apriori.

Block Structure. StruQL's block structure is useful in complex queries, especially in queries that handle optional parts of input data and in queries that integrate multiple sources. For example, the following nested query handles optional `"conference"` links:

```
where Root{r}, r -> "pub" -> p,
      p -> "author" -> a, r -> "title" -> t
link  HomePage(a) ->{"name" -> a, "entry" -> PubEntry(a, p)},
      PubEntry(a, p) -> "title" -> t
{ where p -> "conference" -> c
  link  PubEntry(a, p) -> "publishedIn" -> ConferencePage(c),
        ConferencePage(c) -> "author" -> HomePage(a)
}
```

For every publication `p` matching the first `where` clause, a node with OID `PubEntry(a,p)` is created, together with a `"title"` edge. In addition, if that publication has a `"conference"` link, then `PubEntry(a,p)` has an additional link, `"publishedIn"`, and an `"author"` link is created from the conference page back to the home page. Note that this is *not* equivalent to the flattened query:

```
where Root{r}, r -> "pub" -> p, p -> "author" -> a,
      r -> "title" -> t, p -> "conference" -> c
link  HomePage(a) -> {"name" -> a, "entry" -> PubEntry(a, p)},
      PubEntry(a, p) -> {"title" -> t,
                         "publishedIn" -> ConferencePage(c)},
      ConferencePage(c) -> "author" -> HomePage(a)
```

because nodes `PubEntry(a,p)` are created only for publications having a `"conference"` link.

Blocks are also useful in data integration. The following query integrates authors from `Bib` with employees from `Employee`:

```
{ where BibRoot{r}, r -> "pub" -> p,   p -> "author" -> a,
  link  HomePage(a) -> { "name" -> a,  "entry" -> p }
}
{ where EmployeeRoot{r}, r -> "employee" -> e,
        e -> "name" -> n, e -> "office" -> o,
        e -> "phone" -> p
  link  HomePage(n) -> { "name" -> n, "office" -> o,
                         "phone" -> p } }
```

Persons occurring only in the `Bib` database will have home pages with only two links. For example:

```
HomePage("Joyce") -> { "name" -> "Joyce", "entry" -> p288 }
```

Persons occurring only in the `Employee` database will have home pages with three links. For example:

```
HomePage("Smith") -> { "name" -> "Smith",
                       "office" -> a2345, "phone" -> "x23456"}
```

Finally, persons occurring in both will have four links. For example:

```
HomePage("James") -> { "name" -> "James", "entry" -> p224,
                       "office" -> a52,  "phone" -> "x76543" }
```

Blocks form a tree structure. Sibling blocks do not share any variables except those of their common ancestors. Nested blocks follow the usual conventions. The inner block can introduce new variables and/or new conditions, and inherits all variables of the outer block. For each binding of the variables in the outer block, the inner block is evaluated separately. Nested blocks can always be flattened. For example a STRUQL query of the form:

```
where Pred1(x1, x2, ...)
link  Edges1(x1, x2, ...)
 { where Pred2(x1, x2, ..., y1, y2, ...)
   link Edges2(x1, x2, ..., y1, y2, ...)
 }
```

is equivalent to:

```
{ where Pred1(x1, x2, ...)
  link  Edges1(x1, x2, ...)
}
{ where Pred1(x1, x2, ...), Pred2(x1, x2, ..., y1, y2, ...)
  link  Edges2(x1, x2, ..., y1, y2, ...)
}
```

Sibling blocks can be evaluated in any order; their union defines the output graph. A STRUQL query is deterministic, i.e., its meaning is independent of the evaluation order of any block. Next, we define STRUQL semantics formally.

3.1 Semantics

STRUQL's semantics is described in two stages, which correspond to the query and construction parts of a STRUQL query. We are given a query and an input graph; the result is an output graph. Consider first a query with one block:

```
where   Predicate(x1, ..., xk)
link    Edges(x1, ..., xk)
collect Collections(x1, ..., xk)
```

Here `x1, ..., xk` are all (node and/or label) variables mentioned in any of the three clauses.

In the first stage, the `where`-clause is computed on the input graph and results in a relation $R(x_1, \ldots, x_k)$, with one column for every variable in the query. Let D, called the *active domain*, be the set of all node identifiers, atomic values, and labels occurring in the input graph. Then R consists of all k-tuples $(a_1, \ldots, a_k)$ in D^k which satisfy the predicate in the `where` clause: $Predicate(a_1, \ldots, a_k)$. This is called an *active-domain semantics*, and requires no extra conditions on the query. For example, the following query is well defined:

```
where not(Root{x})
link ...
```

$R(x)$ consists of all node identifiers, atomic values, and labels, except the root's identifier. This query however is domain dependent: if we add more constants to the domain D, its meaning changes. In practice, the system enforces all variables to be *range restricted*, as follows. A variable x occurring in a collection expression $C(x)$ is range-restricted; if x is range restricted, and $x \to R \to y$ is a path expression, then y is also range restricted. Thus, all examples in Sec. 3 are range restricted, while the query above is not. Range-restricted queries are more efficiently evaluated, for obvious reasons. They are also *domain independent*: their semantics is defined as above, but does not change if we replace D with some $D' \supseteq D$.

In the second stage, the `link` and `collect` clauses are computed, as follows. For each row $(a_1, \ldots, a_k)$ in R, we generate all edges and collection memberships in the two clauses. We denote $\bar{x} = (x_1, \ldots, x_k)$, $\bar{a} = (a_1, \ldots, a_k)$, and for some expression E, we denote $E[\bar{a}/\bar{x}]$ the result of substituting each x_i with a_i, $i = 1, k$. Then for each link expression:

```
link       SkolemTerm -> L -> Term
```

the edge $SkolemTerm[\bar{a}/\bar{x}] \to L[\bar{a}/\bar{x}] \to Term[\bar{a}/\bar{x}]$ is added to the output graph. Similarly, for each expression:

```
collect   CollectionName(Term)
```

the value $Term(\bar{a}/\bar{x})$ is added to *CollectionName*.

Finally, we explain the semantics of nested blocks. Based on our previous discussion, nested blocks can be flattened, hence a query has the form:

```
Query :- {Block1} {Block2} ... {Blockp}
```

where each block has no other nested blocks.

The semantics is the following. On a given input graph, each independent block evaluates to some output graph. Then the entire query evaluates to the union of all output graphs; the union is not disjoint, but consists of the union of all nodes, edges, and collection assertions.

3.2 Expressive power and complexity

In this section, we compare StruQL's expressive power with first-order logic (FO) and FO extended with transitive closure and present formal proofs of StruQL's expressiveness. This section is not necessary to understand the rest of the paper, and the reader can continue at Section 3.3, should he choose.

A	B
u	f
e	g
f	k
f	e
g	v

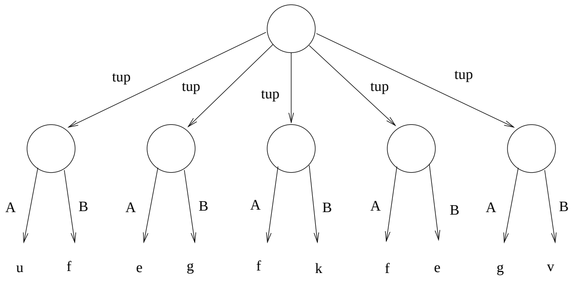

Fig. 4 Encoding of a binary relation as a tree.

STRUQL queries as defined in Fig. 3 are not closed under composition. Here, we study the expressive power of STRUQL's closure under composition. We assume a vocabulary given by a ternary relation $E(x, l, y)$, representing the input graph as a set of triples $(oid, label, oid)$, and a unary relation $Root(x)$, identifying the graph's unique root. In STRUDEL, queries can be composed as follows. The result graph of some query $Q1$ is written to a file. Then query $Q2$ is given that file as input, and its result is written to a different file. This implements the composed query $Q2 \circ Q1$. Some applications use this method to construct more complex graphs in a modular fashion. For this discussion, however, we extend STRUQL's grammar to allow composed queries:

```
ComposedQuery :- Query
               | input {ComposedQuery} Query
```

With that, $Q2 \circ Q1$ would be written as

```
input { where   Predicate1   /* query Q1 */
        link    ...
        collect ...}
where   Predicate2   /* query Q2 */
link    ...
collect ...
```

We illustrate with the following example. Consider a binary relation $R(A, B)$, and two constants u, and v occurring in R. The *accessibility problem* asks whether (u, v) is in the transitive closure of R. To express this in STRUQL's closure we follow established practice and encode (binary) relations as trees: see Fig. 4 for an illustration of this encoding. The query is:

```
input { where   Root{x}, x -> "tup" -> y,
                y -> "A" -> a, y -> "B" -> b
        link    F(a) -> "edge" -> F(b)
        collect U{F("u")}, V{F("v")}
      }
where U{x}, V{y}, x -> * -> y
```

The composed query creates no output graph, but returns *true* or *false* depending whether `("u", "v")` is in the transitive closure. The main idea is that the first query constructs a graph materializing the graph encoded by R. For the example in Fig. 4 the graph is:

```
F("u") -> "edge" -> F("f")    F("e") -> "edge" -> F("g")
F("f") -> "edge" -> F("k")    F("f") -> "edge" -> F("e")
F("g") -> "edge" -> F("v")
```

Proposition 1 *The accessibility problem cannot be expressed by a single* StruQL *query, i.e., without query composition. Hence* StruQL *queries are not closed under composition and require an explicit composition operator.*

Proof Consider any boolean StruQL query Q, which has `where` clauses, but no `link` or `collect` clauses. We will show that, over input trees encoding binary relation instances of $R(A, B)$ (like in Fig. 4), Q is equivalent to a FO sentence over vocabulary $E(x, l, y), Root(x)$: then we will use the fact that FO cannot express transitive closure.

Although Q can have a block structure, we can flatten the blocks and express Q as a union of block-free queries, and prove that each is equivalent to a FO sentence. Thus, we assume that Q is block-free, i.e. consists of a single `where` clause. We show how to translate each condition in the `where` clause into FO. First boolean conditions are translated immediately. Path expressions of the form `x -> L -> y` become $E(x, L, y)$. So it remains to translate only path expressions of the form: `x -> R -> y`, where R is a regular path expression. The important observation is that, on the restricted class of input graphs, there are only six paths: ϵ, `"tup"`, `"A"`, `"B"`, `"tup"."A"`, `"tup"."B"`. We denote these $p_1, \ldots, p_6$. Then the path expression $x \to R \to y$ is translated into $\varphi_1 \vee \ldots \vee \varphi_6$, where each φ_i is as follows. If path p_i does not belong to the regular expression R, then $\varphi_i \equiv false$. Otherwise φ_i is:

$$\varphi_1 \equiv x = y$$
$$\varphi_2 \equiv E(x, \text{"}tup\text{"}, y) \text{ and similarly } \varphi_3, \varphi_4$$
$$\varphi_5 \equiv \exists z.(E(x, \text{"}tup\text{"}, z), E(z, \text{"}A\text{"}, y)) \text{ and similarly } \varphi_6$$

Next, we show that StruQL's closure under composition is as expressive power as first order logic with transitive closure, FO+TC. This language, introduced by Immerman [23], extends first order logic with formulas of the form $TC(\lambda\bar{x}, \bar{x}'.\varphi(\bar{x}, \bar{x}'))$. Here $\varphi(\bar{x}, \bar{x}')$ is any formula in FO+TC denoting a binary relation on k tuples (we assume both $\bar{x}$ and $\bar{x}'$ are k-tuples). Then $TC(\lambda\bar{x}, \bar{x}'.\varphi(\bar{x}, \bar{x}'))$ denotes the transitive closure of φ. Immerman showed that over ordered structures, FO+TC can express precisely the queries in NLOGSPACE[1]. In our context, the structures are over the vo-

[1] This is the class of boolean functions computable by a Turing machine with $O(\log n)$ space. The inclusion NLOGSPACE$\subseteq$PTIME is trivial, and it is open whether it is strict or not.

cabulary $E(x, l, y)$, $Root(x)$ and are unordered. We establish the following elegant result.

Theorem 1 *The closure of* STRUQL *under composition expresses precisely the boolean queries expressible in FO+TC.*

Proof We show first that STRUQL queries can be translated into FO+TC over the vocabulary $E(x, l, y)$, $Root(x)$. Indeed, a `where` clauses with a predicate $P(x_1, \ldots, x_k)$ can be translated into a formula $\varphi(x_1, \ldots, x_k)$, because every regular path expression can be restated in FO+TC. A minor difficulty is that an intermediate graph consists of non-uniform edges: e.g., edges of the form $F_1(x) \rightarrow "a" \rightarrow F_2(y, z)$ and $F_3(x, y) \rightarrow l \rightarrow F_2(y, z)$. We encode the intermediate graph as a $2k + 1$-ary formula $\varphi(\bar{x}, l, \bar{x}')$, using standard padding techniques [2]. Namely, we pick two distinct values $u \neq v$, and encode each node in the new graph as a tuple of arity k, where k is the maximum arity of any Skolem function plus the number p of Skolem functions. More precisely, a node $F_i(x_1, x_2, \ldots, x_l)$ will be encoded as the k-tuple:

$$(x_1, x_2, \ldots, x_l, u, u, \ldots, u, \underbrace{v, \ldots, v}_{p-i})$$

We use the fact that FO+TC is closed under composition.

For the other direction, we prove by induction on the structure of a formula $\varphi(\bar{x})$ in FO+TC that there exists a (possible composed) *StruQL* query $q_\varphi = (\mathsf{where}\ P(\bar{x}'))$, or $q = (\mathsf{input}\ q' \mathsf{where}\ P(\bar{x}'))$, with a superset of φ's free variables (i.e. $\bar{x} \subseteq \bar{x}'$), s.t. φ has the same meaning as the projection of q_φ on the variables $\bar{x}$. The base cases, when φ is $E(x, l, y)$ or $Root(x)$ are trivial: $q_\varphi = \mathit{where}\ x \rightarrow l \rightarrow y$, or $q_\varphi = \mathit{where}\ Root\{x\}$. Consider the case $\varphi = \varphi_1 \wedge \varphi_2$. By induction hypothesis we obtain $q_{\varphi_1} = (\mathit{input}\ q1' \mathit{where}\ p1)$ and $q_{\varphi_2} = (\mathit{input}\ q2' \mathit{where}\ p2)$. Then $q_\varphi = \mathit{input}\ \{q1'\}\{q2'\} \mathit{where}\ p1, p2$. We assume here that the graphs constructed by $q1'$ and $q2'$ have distinct Skolem function names (we can always rename them), hence it is safe to take their union. The case $\varphi = \exists x.\varphi$ is trivial: we take $q_\varphi = q_{\varphi'}$. We do not need to consider $\varphi_1 \vee \varphi_2$ or $\forall x.\varphi$, since these can be expressed by negation. We show next $\varphi = \neg\varphi'(x_1, \ldots, x_k)$. The only way we can express negation is STRUQL is through collections, e.g., $\mathsf{not}\ C\{x\}$. Hence we write a query constructing new nodes of the form $F(x_1, \ldots, x_k)$, with F a new Skolem function name, and insert all those satisfying φ in some new collection C. Then we apply negation to the collection. This requires a composition of two queries; details omitted. Finally, $TC(\lambda\bar{x}, \bar{x}').\varphi(\bar{x}, \bar{x}')$ can be simulated as a composition of two queries, as shown above.

The proof of Theorem 1 relies on the fact that composed queries are, in turn, closed under union (this is needed in the construction of $q_{\varphi_1 \wedge \varphi_2}$ and $q_{\neg\varphi}$). We show next that they indeed are. Syntactically, we cannot write the union of two composed queries as:

```
{ input q1  where p1  link  e1 }
{ input q2  where p2  link  e2 }
```

(The grammar at the beginning of this section does not allow that.) Assuming however that the Skolem function names and new collection names in `q1` and `q2` are disjoint, the query above is equivalent to:

```
input { q1 } { q2 }
{ where p1 link e1 }
{ where p2 link e2 }
```

That is, we compute the union of the input graphs first, then run the two blocks independently on the union graph. Of course, we now have to process {`q1`} {`q2`} recursively, since they may be composed queries too. We stop when both queries `q1, q2` have no composition. It only remains to show the mixed case, when one of the queries does not have composition, while the other one has. This is trivial, since we can always transform a query `where p link e` into `input {} where p link e`: the first query is the empty query (no `where`, `link`, `collect` clauses), and returns the input graph unchanged.

From Theorem 1, it follows immediately that all queries in StruQL and all queries in StruQL's closure under composition have NLOGSPACE data complexity.

3.3 Example Web site

The following example shows how one author's homepage is generated using Strudel[2]. In the remainder of the paper, we refer to this example when describing Strudel's template language and our algorithm for verifying integrity constraints. The main source of data for this homepage is the author's bibliography file. The homepage site has four types of pages: the root page contains general information; an "abstracts" page contains all paper abstracts; and "year" and "category" pages contains summaries of papers published in a particular year or category, respectively. Figure 2 contains a fragment of the site's data graph and was generated by a BibTeX to XML wrapper.

The site graph for the example homepage is defined by the query in Fig. 5. The first clause creates the objects `RootPage` and `AbstractsPage` and creates a link between them. The `collect` expression on line 3 puts the `RootPage` object in a collection with the same name; this is a common idiom so the collection name can be eliminated as it is for `AbstractPage`. For each object x reachable from the root object r by a path labelled `bibliography.article` or `bibliography.inproceedings`, the clause on lines 5–10 collects such x objects in `PaperPresentation`. This object contains the publication's information that will appear in different parts of the site. The link clause also encodes inter-page structure. The first nested clause (lines 14–16) links the general abstracts page to each publication `x`

[2] We encourage the reader to visit the Strudel-generated sites at `http://www.research.att.com/~{mff,suciu}` and `http://www.cs.washington.edu/homes/alon/`.

```
1    // Create Root & Abstracts page and link them
2    link RootPage() -> "AbstractsPage" -> AbstractsPage()
3    collect RootPage{RootPage()}, AbstractsPage()
4    {
5     // Select all conference and journal publications
6     where XMLRoot{r}, r -> "bibliography" -> z, z -> pubtype -> x,
7           pubtype in {"inproceedings", "article"}
8
9     // Collect publications in PaperPresentation collection
10    collect PaperPresentation{x}
11
12    // Link AbstractsPage to papers that contain abstracts
13    { where l = "abstract"
14      link  AbstractsPage() -> "Paper" -> x
15    }
16
17    // Put authors in Author collection
18    { where l = "author"
19      collect Author{v}
20    }
21
22    { // Create a page for every year
23     where  l = "year"
24     link   YearPage(v) -> "Year" -> v
25            YearPage(v)->"Paper"->x,
26
27            // Link root page to each year page
28            RootPage() -> "YearPage" -> YearPage(v)
29    }
30
31    { // Create a page for every category
32     where  l = "category"
33     link   CategoryPage(v) -> "Name" -> v,
34            CategoryPage(v)->"Paper"->x,
35
36            // Link root page to each category page
37            RootPage() -> "CategoryPage" -> CategoryPage(v)
38    }
39   }
```

Fig. 5 Site-definition query for example homepage site

that has an `"abstract"` attribute, and the second nested clause (lines 17–20) puts all objects reachable by an `author` attribute into the `Author` collection.

The third nested `where` clause (lines 22–29) creates a page for each year associated with a publication; the link clause associates each publication object with its corresponding `YearPage`. Lastly, the root page is linked to each year page. A similar clause (lines 31–39) creates a page for each pub-

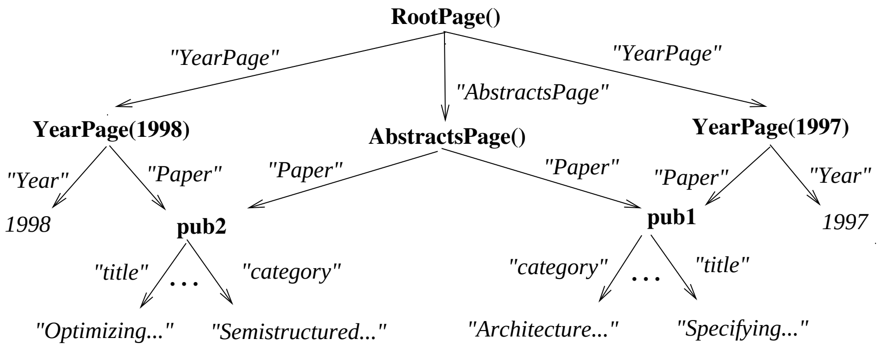

Fig. 6 Fragment of site graph for example homepage site

lication category and links category pages to `PaperPresentation` objects. Note that only one *one* `YearPage` object will be created for each distinct value of `v` (similarly for `CategoryPage`), thus all publications in the same year (or category) will be grouped together.

Fig. 6 depicts a fragment of the generated site graph; for clarity, it excludes the result of the last nested clause that produces category pages. Note that the site graph encodes both the site's content and its structure. For example, the `YearPage` objects have links to year values and to their associated papers. All leaf objects contain page content, e.g., the titles of publications. Declarative specification of the site graph is powerful, because the site builder can specify its structure in any order he chooses. For example, he can define the pages "top down" from the root, or first define each group of related pages and then link them.

4 Template Language

One premise of STRUDEL's design is that a site's HTML rendering is separable from the site's content and structure. STRUDEL's template language allows the user to specify a site's HTML rendering. A template is a function that maps an object in a site graph to HTML text. The template's expressions produce HTML, which are concatenated to produce its result. Figure 7 contains the EBNF grammar for the template language. Figure 8 contains the templates for the example home-page site.

HTML text, the format expression (`sfmt`), conditional expression (`sif`), and enumeration expression (`sfor`), are sufficient for emitting a site graph in HTML. An attribute expression, e.g., `YearPage.Year`, denotes the set of objects reachable by edges labeled with the given attributes. An attribute expression implicitly refers to the template's object argument, named `this`, but can refer explicitly to any object variable, e.g., `@this.YearPage.Year`. Sometimes more general computation is necessary during HTML generation; the `sjava` construct provides an "escape" into the Java programming language, which permits the evaluation of arbitrary Java code.

$$
\begin{array}{lll}
\mathit{Template} & :- & \big\{\mathit{Body}\big\} \\
\mathit{Body} & :- & \mathit{HTML} \\
 & \big| & <\textbf{sfmt}\ \mathit{AttrExpr}\big[\textbf{link} = (\mathit{AttrExpr}\big|\mathit{String})\big]> \\
 & \big| & <\textbf{sif}\ \mathit{CondExpr}>\mathit{Body}</\textbf{sif}> \\
 & \big| & <\textbf{sfor}\ \mathit{ObjVar}\ \textbf{in}\ \mathit{AttrExpr} \\
 & & \big[\textbf{order} = (\textbf{ascend}\big|\textbf{descend})\ \textbf{key} = \mathit{AttrExpr}\big]>\mathit{Body}</\textbf{sfor}> \\
 & \big| & <\textbf{sjava}>\mathit{JavaCode}</\textbf{sjava}> \\
\mathit{AttrExpr} & :- & \big[@\mathit{ObjVar}\textbf{.}\big]\mathit{Attribute}\big\{\textbf{.}\mathit{Attribute}\big\}
\end{array}
$$

Fig. 7 EBNF Grammar for HTML-Template Language

For each object in a site graph, STRUDEL's generator applies the appropriate template to the object to produce its HTML value. Each object in a site graph has a user-specified *generation mode*: *page* or *page component*; all leaf objects, i.e., atomic values, are page components. In the example site, all objects in the `RootPage`, `AbstractsPage`, `YearPage`, and `CategoryPage` collections are realized as pages; objects in the `PagePresentation` collection are realized as page components.

The format expression (`sfmt`) maps an object, denoted by an attribute expression, to HTML. An attribute expression is either a single attribute, e.g., `Name`, or a bounded sequence of attributes, e.g., `YearPage.Year`. that reference a set of objects reachable by edges labeled with the given attributes. In the `YearPage` template (Fig.8), <sfmt Year>, refers to the atomic value reachable by the attribute expression `@this.Year`, and is replaced by its integer value rendered in HTML. Format expressions are concise, because the generator uses type-specific rules to determine an object's rendering in HTML. For most atomic values (e.g., integers), the object's value is converted to a string in HTML. For some atomic values, e.g., those with mime-content type PostScript or URL, the generator produces a link to its value. For example, in the `PageFormat` template, <sfmt postscript link=title> is replaced by a link to the object's `postscript` attribute, which is a PostScript file, and the object's `title` attribute is emitted as the link's tag text.

A complex object's generation mode determines how it is formatted. In the `PaperPresentation`'s template, <sfmt author>, always refers to an `Author` object `a`, which is a page component, so it is replaced by `a`'s value in HTML, but the expression <sfmt AbstractPage link="Abstract"> refers to a `AbstractPage` object, which is a page, so it is replaced by a link to the appropriate page; the `link` directive specifies the link's tag text.

The `sif` expression evaluates a conditional expression and then evaluates the appropriate branch. For example, in the `PaperPresentation` template:

```
<sif booktitle> <sfmt booktitle> <selse> <sfmt journal> </sif>
```

RootPage template:

```
<html>
<!-- Plain HTML text omitted -->
<h2><sfmt AbstractsPage link="All Paper Abstracts"></h2>
<h2>Publications by Year</h2>
<UL>
<sfor p in YearPage order=ascend key=Year>
  <sfmt @p link=@p.Year><br>
</sfor>
</UL>
<h2>Publications by Subject</h2>
<sfmt CategoryPage link=Name UL order=ascend key=Name>
<!-- More plain HTML text omitted -->
</html>
```

YearPage template:

```
<html><h1>Publications from <sfmt Year></h1>
<UL><sfor p in Paper> <LI><sfmt @p> </sfor></UL></html>
```

CategoryPage template:

```
<html><h1>Publications on <sfmt Name></h1><sfmt Paper UL></html>
```

AbstractsPage template:

```
<html><H1>Paper Abstracts</H1><sfmt AbstractPage embed UL></html>
```

Author template:

```
<sfmt lastname>
<sif firstname>, <sfmt firstname>
<selse><sif initials>, <sfmt initials>.</sif>
</sif>
```

PaperPresentation template:

```
<i><sif postscript><sfmt postscript link=title>
   <selse><sfmt title>
   </sif></i>.
By <sfmt author enum delim=", ">.
<i> <sif booktitle> <sfmt booktitle>
    <selse> <sfmt journal>
    </sif>, <sfmt year>.</i> <br>
<sif><sfmt abstract embed><br></sif>
```

Fig. 8 HTML Templates for example homepage site

tests for a `booktitle` attribute and emits its value if it exists, otherwise it emits the `journal` attribute's value. Predicates in conditional expressions include builtin relational operations as well as user-defined external predicates, written in Java.

Objects can have multiple instances of the same attribute, e.g., YearPage objects have multiple Paper attributes. The sfor expression binds an object variable to each object denoted by its attribute expression and evaluates its body for each binding. In the RootPage template,

```
<UL>
<sfor p in YearPage order=descend key=Year>
  <sfmt @p link=@p.Year><br>
</sfor>
</UL>
```

binds the object variable p to each value of the expression this.YearPage and emits p's value. The order directive sorts objects in either lexicographically increasing or decreasing order; if the objects are complex, the optional key value specifies the attribute that should be used as the key. The sfor expression above orders the YearPage objects in descending order by their Year attribute.

Enumerating all values of an object's attribute is common, so we abbreviate common idioms. For example the expression

```
<sfmt YearPage link=Year UL order=ascend key=Year>
```

is equivalent to the last example above. Attributes are often emitted in ordered and unordered lists. For example, <sfmt AbstractPage embed UL> in the AbstractsPage template is shorthand for:

```
<UL><sfor a in AbstractPage><LI><sfmt @a embed></sfor></UL>
```

We describe other template languages and evaluate the necessity of a special-purpose templating language in Section 6.

5 Verifying Integrity Constraints

Unlike a site implementation in an imperative scripting language, site-definition queries in STRUQL can be analyzed. For example, we can analyze a STRUQL query to verify that the Web site it defines satisfies certain constraints. In this section, we formally define the constraint verification problem. To motivate the problem, consider the following integrity constraints that we may wish to enforce on the Web site generated by our example query.

1. All PaperPresentation pages are reachable by a path from RootPage.
2. If a publication's PostScript source exists, then its PresentationPage is linked to it.
3. Unless a link labeled "Back to Regular Site" is followed, no page reachable from "TextOnlyRoot" contains images.

We define the verification problem as an entailment problem of a query 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 $\exists$.

Given a labeled, directed data 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 1 The following sentences represent the three examples above.

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

To simplify presentation, the predicate $PaperPresentation(X)$ is equivalent to the conditions on lines 6–7 in Fig. 5:

```
XMLRoot{r}, r -> "bibliography" -> z, z-> pubtype -> X,
pubtype in {"inproceedings", "article"}
```

5.1 Verification algorithm

Next, 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, not 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 1, Q_1, Q_2 refer to the site graph; in (2), Q_1 refers to the data graph.[3] In the former cases, we need to consider

[3] 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.

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.

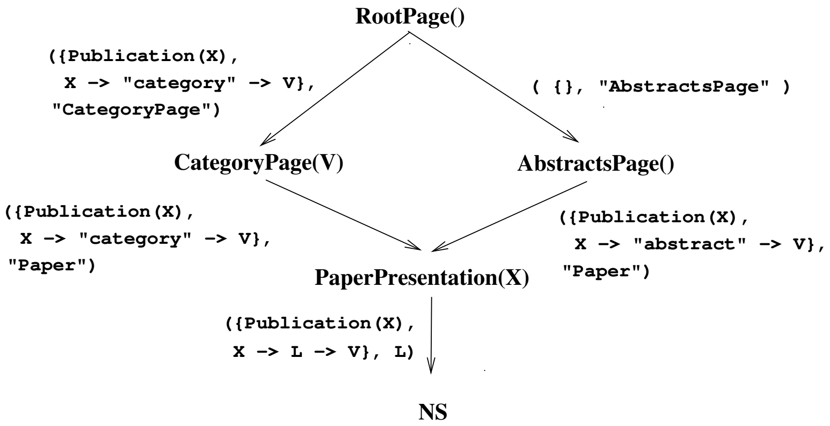

Fig. 9 Fragment of site schema for example homepage site.

To perform the translation, we use a novel data structure, the *site schema*, that provides an abstract graphical representation of a STRUQL expression. Due to space limitations, we consider only a simplified form of site schema. A fragment of the site schema for the homepage Web site is shown in Figure 9. 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, 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 2 In our example, the following formula describes the condition for existence of a path from `RootPage()` to `PaperPresentation(X)`:

$$(Publication(X) \wedge X \rightarrow \text{"}category\text{"} \rightarrow v) \vee$$
$$(Publication(X) \wedge X \rightarrow \text{"}abstract\text{"} \rightarrow v)$$

The first disjunct describes a path through `CategoryPage(V)`, and the second describes a path through `AbstractsPage()`. 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:

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

Suppose we want to write a condition that expresses the existence of a path from `RootPage()` to `PaperPresentation(X)` that *does not* go through `AbstractsPage`. In this case, we only consider paths in the site schema that do not go through `AbstractsPage`, therefore the condition is simply:

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

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 2 *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 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 3 *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 2 and 3 combined capture many cases encountered in practice for which the resulting algorithm can be implemented relatively efficiently.

The proof of Theorem 2 proceeds by reducing the verification problem to a logical entailment problem for STRUQL-query expressions, which is known to be decidable [17]; the case for simple regular expressions has been shown to be in NP. The proof of Theorem 3 proceeds by a reduction to the problem of entailing a datalog expression from a non-recursive datalog expression, which has shown to be decidable in [9].

6 Related Systems

Many commercial systems exist for designing and implementing Web sites. Here, we describe a variety of systems whose design goals, like Strudel, include isolating the orthogonal tasks of Web-site development. We also describe STRUDEL's relationship to emerging XML technologies. For more comprehensive descriptions, we refer the reader to thorough reviews of site-development tools [16,20].

6.1 Model-driven design systems

Many of the problems associated with designing a Web site, such as modeling the site's content, specifying navigational structure, and customizing visual presentation, have been studied in the context of hypermedia systems, and many of the solutions to these problems for hypermedia systems are transferable to Web-site design. Several research systems, Autoweb [25], OOHDM [27], and Araneus [6] ascribe to a top-down methodology of Web-site design, whose purpose is to isolate the orthogonal tasks of site design and codify each in a meta-schema. A "conceptual" design, i.e., an abstract model of the site, is produced first and can be specified in an Entity-Relationship schema (e.g., Autoweb and Araneus) or in an object-oriented model (e.g., OOHDM-Web). The "navigational" design, i.e., how the user can move between entities in the conceptual design, is specified next and is often specified as a declarative view over the conceptual design. The "presentation" design specifies how entities in the conceptual and navigational designs are presented visually to the user. Finally, the "application" or physical design specifies the relationship between the higher-level designs and the underlying application's databases.

Although the general methodology is the same, each system provides different tools, with varying levels of automation, to implement a design. The

Autoweb system provides one tool to automate each step and because of its strict adherence to the design methodology, requires the site implementor to use specific tools. One limitation is that Autoweb does not support intensional definition of site content, which can result in redundant definitions of related components, nor does it support querying of its meta schemas. The Araneus data model (ADM) supports an intensional description of a Web site as a graph of strictly typed *page schemes*. Its query language (ULIXES) defines a relational view of an ADM graph; multiple data sources are integrated by relational queries over these relational views. A second query language (PENELOPE) transforms an integrated, relational view back into an ADM graph; a final step renders an ADM graph as a browsable site. The OOHDM-Web system only partially automates translation of design schemas into the scripting language CGI-Lua; the user is required to implement the rest by hand. We note that as an implementation tool, STRUDEL is complementary to site-design tools, because its declarative query language is well-suited to automatic generation and could be used as an implementation language for a variety of design systems.

6.2 Server-side scripting languages

Server-side scripting languages include Embperl, PHP, Netscape's server-side Javascript, Sun's Java Server Pages (JSP), Microsoft's Active-server Pages (ASP), and the markup pages of Allaire's Cold Fusion. A common goal of these languages is to eliminate the details of CGI-scripting and simplify the tedious development of Web applications in languages like Perl, which provide few high-level programming constructs and result in code that is hard to modify and reuse.

Server-side scripts are typically plain HTML text interleaved with segments of program code that are interpreted by the server. All the languages are imperative, and most provide high-level features to simplify development, such as session tracking and management, access to stored objects (e.g., in Java or Active-X), and read-only and transactional access to databases. Overall, these languages increase a Web developer's "stickiness" to a particular vendor, because scripts must be interpreted by the vendor's Web server. Some tools also include "wizard" or rapid application development (RAD) environments, which provide a drag-and-drop design interface and that generate code in the underlying scripting language. Although these languages and environments have improved significantly the process of Web-site development, a site definition is still comprised of disparate scripts that interleave presentation with content. Extracting a holistic definition of the site's content and structure from scripts would be difficult, and therefore any analysis or optimization of the implementation equally difficult.

In contrast, STRUDEL separates the intensional, declarative definition of a site from its presentation. STRUQL is closer in spirit to query languages for XML, and STRUDEL's template language more closely resembles a stylesheet language, like XSLT [7], than it does a server-side script. We discuss this relationship next.

6.3 XML technologies

XML, XSLT, and several XML query languages are already influencing Web-site development. In particular, XML and XSLT decouple page content from page presentation, which makes it possible for applications other than browsers to process page content. Although STRUDEL predates XML, its data model, query language, and template language are so similar to XML, several query languages for XML, and XSLT, that the translation from STRUDEL into these more widely used languages can be automated completely.

Clearly, an individual page or an entire site can be represented in XML; STRUDEL already emits a site's contents, i.e., a site graph, as an XML document. In lieu of STRUQL, an XML query language, such as XML-QL [10] or YaTL [8], could be used to declaratively define a site. STRUQL's semantics are so closely related to that of XML-QL that the first implementation of XML-QL translated queries into STRUQL for evaluation. In addition, STRUDEL's templates can be translated into a subset of XSLT. Each STRUDEL template file is equivalent to one `<xsl:template>` rule. The template format expression `<sfmt ...>` translates into the XSL expression `<xsl:value-of select="...">`; for example, `<sfmt Name.Last>` is equivalent to `<xsl:value-of select="Name/Last">`. Similarly, the conditional expression `<sif ...>` translates into `<xsl:choose ...>`, and the iteration expression `<sfor ...>` into `<xsl:for-each ...>`. Unlike XSLT, expressions in STRUDEL's template language are guaranteed to terminate (XSLT can express non-terminating programs), and STRUDEL can render more than one page at a time, whereas an XSLT stylesheet defines only one page. An important benefit of XSLT is that pages can be rendered on the client or server. We expect that even though XSLT and existing XML query languages are oriented towards individual documents, they could easily be extended to define complete Web sites and could provide the benefits of STRUDEL, plus many more, to a wider audience.

6.4 Other related languages

Several systems are inspired by database research on semistructured data models and novel techniques for data integration. WebOQL [4] supports querying of existing Web sites and can produce views of sites as restructured graphs. Like STRUDEL, WebOQL provides a uniform, semistructured data model (called a *hypertree*), and its query language supports regular path expressions, can restructure graphs, and is compositional; unlike STRUDEL, its data model supports records and ordering. Also, WebOQL expresses the HTML rendering of pages in queries.

YAT [8] is a semistructured database-management system intended primarily for translation and integration of data in heterogeneous data sources. The YAT data model supports ordered labeled trees and its query language, YaTL, is a rule-based tree rewriting language. The body of a rule contains patterns and predicates to filter the input trees. The head of a rule contains a single pattern that describes how to restructure the data filtered by the body. YaTL uses Skolem terms to manipulate identifiers and create complex graphs. YaTL's innovation is its support for ordered lists and bags. YAT also has been applied to Web-site management. A YaTL program can restructure raw data as a tree and produce as a result a tree that corresponds to the abstract syntax tree of the desired HTML pages. YAT's main disadvantage is that it does not clearly separate between Web-site structure, content and graphical presentation.

STRUDEL is influenced by other domain-specific languages for implementing Web-based applications. One example is MAWL [5], a device-independent language for programming form-based services, which can be realized as Web applications or as interactive voice-response systems. MAWL provides one high-level language for specifying the interactions between the user and the underlying application and a second template-language for specifying presentation. Although STRUDEL's application is different, its separation of application logic from presentation and its template language are closely related to MAWL.

7 STRUDEL Descendents

STRUDEL has three descendents, each of which extends and improves upon STRUDEL's original ideas of task separation and declarative specification. We describe each here.

7.1 Strudel-R.

Given a declarative specification of a Web site over a large, relational database, the STRUDEL-R [18] system addresses the problem of optimizing the run-time generation of the site's pages. Unlike STRUDEL, which assumes that the sites' content is obtained from multiple semi-structured data sources, the STRUDEL-R system[19] assumes that a site's content is derived from a single, large relational database. Like STRUDEL, STRUDEL-R is based on a declarative definition of the site content and structure. STRUDEL-R's goal is to improve the run-time behavior of data-intensive Web sites by taking advantage of the declarative specification. When a site's content is populated from a large database, an important problem is determining when to compute the pages in the site [26] and/or the corresponding nodes in the logical model. One approach is to materialize the site completely, i.e., evaluate all the database queries in the site definition, and compute the complete site before users browse it. A second approach, often employed by commercial tools, is to precompute only the root(s) of the site, and issue a set of parameterized queries to the database when a page is requested. Both approaches have advantages and disadvantages: stale data, space overhead, and no support for dynamic inputs in the former case, and unacceptable or unpredictable wait time for pages in the latter case. In an experimental study [18], we examine the optimal tradeoff between precomputation and dynamic evaluation, propose several techniques for optimizing the run-time behavior of Web sites, and describe a framework for automatically compiling site specifications into run-time policies that incorporate these optimizations.

7.2 Functional Strudel.

The Fun-STRUDEL system [14] addresses the software-engineering problem of producing site implementations that have the characteristics of well-designed software systems: i.e., they are reusable, analyzable, and optimizable. These goals were motivated by our effort to apply STRUDEL to an "industrial strength" Web site. Unlike our first applications of STRUDEL (mostly small Web sites and personal home pages), our first production application has several types of complex pages and integrates data from several gigabyte-sized sources (see Section 8). For this site, STRUDEL proved inadequate in two ways. First, specifying a large site in a single, monolithic STRUQL query results in an implementation that is hard to understand, reuse, or extend by multiple people. Second, STRUDEL queries are always evaluated completely (or *eagerly*) and materialize the entire site graph. Eager evaluation was inappropriate for this site, which requires both static (i.e., precomputed) and dynamic (i.e., on-demand) pages.

To address these problems, Fun-STRUDEL extended STRUDEL with *query functions*, which modularize site-definition queries and are the minimal unit of query evaluation, and *declarative forms*, which support dynamic binding of variables. In addition to eager evaluation, Fun-STRUDEL supports *lazy* evaluation of query functions, i.e., at "click time"; a lazily evaluated query produces a *dynamic* site graph. Fun-STRUDEL supports flexible site-generation strategies by combining eager and lazy evaluation of query functions to produce sites that have both static and dynamic parts.

The ability to support multiple site-generation strategies is especially important for data-intensive Web sites, in which the time to produce pages is non-uniform; e.g., some functions may submit expensive queries to an external source. Unlike STRUDEL-R, in which the site implementor has the ability to extend the relational database with precomputed views, in Fun-STRUDEL, we assume the site administrator can only access the underlying sources as "black boxes". Fun-STRUDEL allows the site implementor (or automatic site optimizer) to specify one or more site-evaluation strategies, separately from the site definition. This approach is more flexible than current practice, in which one site-generation strategy (usually fully static or dynamic) is programmed explicitly in the implementation.

7.3 Tiramisu.

The TIRAMISU system [3] addresses one serious limitation of declarative Web-site management systems: the separation of the Web-site design tool from the implementation tools. For example, users of STRUDEL are forced to implement their site only with STRUDEL. In practice, users may want to use specific implementation tools (e.g., visual HTML editors, Active Server Pages), because they are more appropriate for certain tasks or because of organizational requirements.

TIRAMISU [3] provides a declarative site-specification language that is decoupled from specific implementation tools. In TIRAMISU, the site designer defines declaratively the structure and content of the site, and separately specifies the tools that should implement different parts of the site. TIRAMISU's *implementation manager* coordinates the implementation of the site with the tools, ensuring that the different parts of the site fit together seamlessly. Any implementation tool that supports TIRAMISU's common application-program interface can be used in concert with other tools to implement a site.

8 STRUDEL in Practice

To illustrate how STRUDEL is used in practice, we describe an internal AT&T Web site, called "hightoll notifier" (HTN), which is implemented using Fun-STRUDEL. HTN identifies business-customer accounts that appear to be high risk, i.e., ones whose bills may go unpaid. Statistically, customers that have a significant increase in their telephone usage over a short period of time are more likely to not pay their bills than those customers that have constant daily usage. Other high-risk indicators include the customer's credit record and their ability to pay previous bills on time.

The data in the HTN site must be current to within one day or even a few hours, so that account representatives can identify and contact high-risk accounts before the account further increases its usage or goes into arrears at billing time. Before the HTN site existed, account representatives might have waited several weeks before they had sufficient information to identify high-risk accounts. The HTN site is a tremendous success, because it provides in real time an integrated view of high-risk accounts.

HTN is a good example of a data-intensive Web site: it integrates data from multiple sources and allows the site user to "drill down" from the high-level, summary perspective to the low-level source data. HTN computes usage statistics on approximately 250 million phone calls daily and integrates information from several sources: phone-call records, long-term account information, and external credit reports. Of the 1.6 million business accounts tracked, approximately 4500 are identified daily as potential risks. The site has five levels: each subsequent level provides a more detailed view of the high-risk accounts. The *root page* allows an account representative to select the types of high-risk accounts to track, e.g., a particular market segment. The *hot list* page lists the set of accounts in the chosen segment and orders them by a risk metric. The hot-list page points to *account pages*, which displays a summary of an account's usage in textual and graphical form. A *report page* is accessible from several pages in the site and presents the account's risk metrics and allows the account rep to view and record interactions with the customer. The most detailed page presents the original phone-call records from which the usage summary is computed.

The original HTN site was implemented using scripting languages, e.g., Korn shell and Perl, and common Unix command-line tools, e.g., awk, sed, and grep. The scripts process user inputs, invoke Unix tools to handle simple data-management tasks, and format and emit HTML pages. Several C programs implement rudimentary database operations. Although some scripts differentiated the three site-creation tasks, most scripts interleaved them. The result is a loosely related set of scripts that implemented the required functionality, but that have the characteristics of a poorly implemented software system: the code is hard to understand and extend, because the program's tasks are undifferentiated. These problems complicated extension and prevented reuse of HTN's first implementation. We admit these tools do not compare to more advanced site-creation products. We note, however, that many Web sites inside AT&T integrate information from a variety of sources, such as legacy relational databases and flat files, and therefore need general-purpose tools to access and manipulate data.

Given the limitations above, the site was re-engineered completely using Fun-STRUDEL [14] and the Daytona relational database management system [21]. Table 1 compares the two implementations. We compared the total number of files and total number of non-empty, non-comment lines of code for each implementation. Reducing the total line count is not a definitive measure of improvement, but it does indicate the relative effort required for each implementation. Each source-code file was categorized as primarily site-definition code, HTML-template code, or general-purpose Java code. In the Fun-STRUDEL implementation, 66% of the code is devoted to page presentation, but less than 30% is required to define the site. This is encouraging, because the site-definition query contains the potentially reusable part of the specification and is the first and only component that a user would read to understand the site's definition. In the original implementation, 75% of the code is devoted to site definition, but more importantly, the code to access data, to define site structure, and to emit HTML code is interleaved, making it difficult to modify or extend. Overall, the Fun-STRUDEL implementation is 1.6 times smaller than the original implementation, but if we compare only the code for site-definition, it is more than 4 times smaller. Also, the Fun-STRUDEL definition is encapsulated in one file, whereas the original definition was distributed over 23 files.

The STRUDEL implementation has several benefits. One important benefit is that STRUDEL's separation of content and presentation makes it possible to export HTN's data in XML, with no changes to the implementation.

Type of code	Implementations: Fun-STRUDEL # lines	Fun-STRUDEL # files	Original # lines	Original # files
Site definition	**291**	1	**1198**	23
HTML templates	673	11	42	1
Java code	41		392	1
Total	**1005**	12	**1632**	26

Table 1 Comparison of HTN site implementations

This has increased the site's value significantly, because applications other than browsers can use its content. A second benefit was that unlike the original implementation, the Fun-STRUDEL implementation supports flexible site-generation strategies. For example, we implemented some simple strategies, such as precomputing frequently accessed hot lists and report pages. These added only 10 lines to the site-definition query, and in the best cases, reduced page-generation time from 12 seconds to less than 2 seconds. The strategies *extend* the original query with hand-coded optimization rules. Our next challenge is to generate these strategies automatically. HTTP-server trace logs and STRUDEL profiling statistics can provide useful optimization information.

9 Discussion

This work makes several important conceptual and practical contributions. First, we identified Web-site implementation as a data-management problem and recognized that separating the management of a site's data, the specification of its content and structure, and the visual presentation of its pages, facilitates important site-management tasks, such as integrating data from multiple sources, generating multiple views of a site, and enforcing integrity constraints on sites. Our key insight was that these problems are best solved by a declarative query language. Second, we based STRUDEL's architecture on these important ideas and built a prototype system that has been used to implement a variety of Web sites. Third, we have shown that STRUQL is a simple, but expressive query language and that its declarative semantics makes it easy to understand and more importantly, easy to analyze. Finally, our experiences with STRUDEL have led to three distinct, but complementary research efforts, each of which expands on STRUDEL's foundation of task separation and declarative specification.

STRUDEL is not a commercial product, nor does it have a large base of external users, but its prototype is robust enough to use in production on a daily basis. Based on our experience, STRUDEL is well suited to data-intensive Web sites that integrate information from both relational and non-relational data sources. We have found that STRUDEL site definitions are easy to maintain, because whenever the data sources are added or removed, the short STRUQL definitions clearly identify what parts of the site are effected. Similarly, whenever site presentation changes, modifications are isolated to template files. Typically, sites are defined only using STRUDEL, so we cannot comment on how well STRUDEL works with other tools. This problem is addressed by TIRAMISU. Finally, the separation of content and presentation makes STRUDEL sites more valuable than script implementations, because the site's content can be exported in both XML and HTML, with no changes to the implementation.

STRUDEL is not well suited for highly transactional sites that update the underlying store continuously. The HTN site, for example, is primarily read-only, but it does permit some updates to the underlying data sources, which can invalidate pages cached by STRUDEL. STRUQL does not have an update semantics, i.e., a formalism for specifying updates to a query's domain, nor a syntax for specifying updates. Given an update semantics, STRUDEL could support incremental update of a site, i.e., identify those parts of the site graph invalidated by an update and recompute automatically the pages effected. Currently, simple external scripts determine which pages are invalid after an update and must be recomputed by STRUDEL. This makes STRUDEL less appropriate for transactional sites.

Most of STRUDEL's benefits hinge on its declarative query language, but it remains to be seen whether STRUDEL will impact directly common practice. There are several obstacles to wider acceptance of STRUDEL and its ideas. Programmers tend to resist learning a new language, unless overwhelming evidence indicates its benefits far exceed existing solutions. Successful languages, such as Perl, tend to have many grassroots supporters who implement many applications in the language. Moreover, Web-site developers much prefer to use the GUIs of RAD tools, than to write code in a new language. Most STRUDEL users, for example, are not professional Website developers, but are computer scientists who are interested in STRUDEL's semantics or who process and analyze data that they need to browse. Even if STRUDEL were to support a visual notation, we do not expect STRUDEL itself to develop a huge following. We do expect, however, that STRUDEL's ideas can influence the use of XML technologies in site development. Already, the emergence of XSL and declarative XML query languages are promising indicators that many of STRUDEL's ideas can be applied to more mainstream tools.

Acknowledgements STRUDEL exists in large part to the efforts of Jaewoo Kang, Sandra Sudarsky, and Igor Tatarinov.

Availability STRUDEL is available at `http://www.research.att.com/sw/tools/strudel`. Its users' guide is at `http://www.research.att.com/~mff/strudel/doc`.

References

1. S. Abiteboul. Querying semi-structured data. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison Weseley, 1995.
3. C. Anderson, A. Levy, and D. Weld. Declarative web-site management with Tiramisu. In *ACM SIGMOD Workshop on the Web and Databases (WebDB'99)*, Philadelphia, PA, June 1999.
4. G. Arocena and A. Mendelzon. WebOQL: Restructuring documents, databases and webs. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, Orlando, Florida, 1998.
5. D. Atkins, T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a domain specific language for form-based services. In *Proceedings of Conference on Domain-Specific Languages*, pages 37–49, 1998.
6. P. Atzeni, G. Mecca, and P. Merialdo. Design and maintenance of data-intensive web sites. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, pages 436–450, Valencia, Spain, 1998.
7. J. Clark. XSL transformations (XSLT) specification, 1999. `http://www.w3.org/TR/WD-xslt`.
8. S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion. In *Proc. of ACM SIGMOD Conf. on Management of Data*, Seattle, WA, 1998.
9. S. Cosmadakis and P. Kanellakis. Parallel evaluation of recursive rule programs. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Washington, D.C., 1986.
10. A. Deutsch, M. Fernńdez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the Eights International World Wide Web Conference (WWW8)*, Toronto, 1999.
11. M. Fernández, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: experiences with a web-site management system. In *SIGMOD*, Seattle, Wash., June 1998.
12. M. Fernández, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Record*, 26(3):4–11, Sept. 1997.
13. M. Fernández, D. Florescu, A. Levy, and D. Suciu. Verifying integrity constraints on web sites. In *IJCAI*, 1999.
14. M. Fernández, I. Tatarinov, and D. Suciu. Declarative specification of data-intensive web sites. In *USENIX Conference on Domain-Specific Languages*, 1999.
15. D. Florescu, A. Levy, I. Manolesu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proc. of ACM SIGMOD Conf. on Management of Data*, 1999.
16. D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Record*, 27(3), Sept. 1998.
17. D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Seattle,WA, 1998.
18. D. Florescu, A. Levy, D. Suciu, and K. Yagoub. Optimization of the run-time management of data intensive Web-sites. In *Proc. of the 25th VLDB Conference*, Edinburgh, Scotland, Sept 1999.

19. D. Florescu, A. Levy, D. Suciu, and K. Yagoub. Run-time management of data intensive Web-sites. Technical Report RR-3684, INRIA, 1999.
20. P. Fraternali. Tools and approaches for developing data-intensive web applications: a survey. *ACM Computing Surveys*, Sept. 1999.
21. R. Greer. Daytona. *Proceedings of the SIGMOD International Conference on Management of Data*, June 1999.
22. R. Hull. Managing semantic heterogeneity in databases: A theoretical perspective. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 51–61, Tucson, Arizona, 1997.
23. N. Immerman. Languages that capture complexity classes. *SIAM Journal of Computing*, 16:760–778, 1987.
24. J. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.
25. P. Paolini and P. Fraternali. A conceptual model and a tool environment for developing more scalable, dynamic, and customizable web applications. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, 1998.
26. B. Proll, W. Retschitzegger, H. Sighart, and H. Starck. Ready for prime time - pre-generation of Web pages in tiscover. In *ACM SIGMOD Workshop on the Web and Databases (WebDB'99)*, 1999.
27. D. Schwabe and G. Rossi. An object oriented approach to web-based application design. *Theory and Practice of Object Systems, Special Issue on the Internet*, 4(4):207–225, 1998.
28. J. D. Ullman. Information integration using logical views. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.
29. Extensible markup language (XML) 1.0, 1998. `http://www.w3.org/TR/REC-xml`.