SPHINX: Schema Integration by Example[1]

Francois Barbançon and Daniel P. Miranker

Department of Computer Sciences
University of Texas
Austin, TX 78712
12/02/02

Abstract

We focus on the problem of semi-automated query discovery for XML views without requiring the intervention of an expert to guarantee a correct final result. Given multiple independent sources of heterogeneous XML data structures, our tool, SPHINX, lets a non-technical user define views using example-based, graphical, interaction. SPHINX embodies a syntactically derived meta-model of federating view definitions. The meta-model of federating view definitions enables, first, an active-learning method which removes the burden of generating the examples from the user. Second, SPHINX can identify when the learned view definition converges to a vetted, semantically accurate result. Thus, the users do not need to verify the correctness of the transformation.

1. Introduction

Almost any kind of data has become available online using a proliferation of domain-specific XML standards. Yet, cooperation across organizations and their respective databases is still the exception rather than the rule. Widespread adoption of any schema description, (DTD or XML schema), is rare. Proprietary and legacy interests will continue to ensure this into the foreseeable future. The burden of data integration will continue to be on the user of the data rather than the publisher. The task of tapping a set of online data sources usually falls upon a specialist and requires a large effort.

The Query-by-Example (QBE) system established a framework where non-technical users could define SQL queries against relational databases [36]. We seek a similarly facile method to enable non-technical users (such as a market research or financial analyst) to perform local integration of data sources for their own needs. Since it is for their own need, it follows that they understand the semantics of their domain.

It has been established that higher-order query languages provide a powerful and concise means for specifying integrating views (albeit in a complex syntax) [14,33]. A challenge we address is; whereas in a QBE system a single example uniquely determines a first-order query, in a higher-order context a single example usually entails a large number of queries.

Consider a simple problem, drawing data from two sources (shown in Figure 1), where we have two distinct schemas, Product Review and Supplier Catalog, being mapped to a third, target schema. With a QBE-like graphic tool, it is easy to come up with a simple schema match. Several issues remain in order to complete this job. For instance, how identical products between the Product Review database and the Supplier Catalog should be matched in order to populate the target schema (join operation). Another issue is the tag '5-star' in the target schema might indicate only products with '5-star' ratings should populate the target schema, or it might simply be an object name with no particular semantic denotation. These and others kinds of schematic discrepancies cannot always be dealt with in the framework of QBE (or of SQL), or by drawing lines on a diagram.

[1] This work was supported by grants from Telcordia Technology inc. and the Texas Higher Education Coordinating board.

However, we assume that the user, while non-technical, is the owner and/or end consumer of the data, and as such has intuitive knowledge about semantic elements such as composite keys, foreign keys and other integrity constraints. But that same user will have no formal or syntactic knowledge of how to express those semantic elements. Thus our system's goal will not be to deduce or learn those elements by mining available information, but rather to extract them from the user by a simple yet formally powerful active learning interaction.

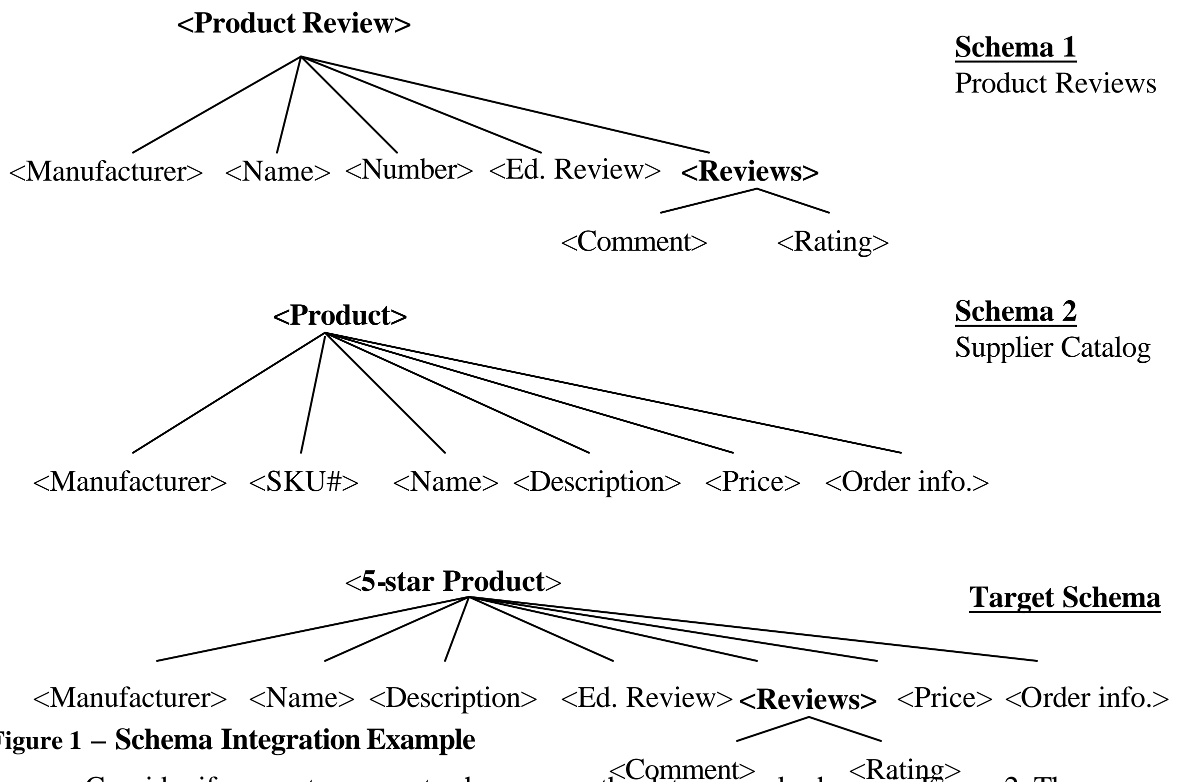


Figure 1 – Schema Integration Example

Consider if our system were to show a user the data example shown in Figure 2. The example comprises a data instance from each schema, and proposes a mapping of this data to populate an instance of the target schema. This example constitutes what could be termed a data mapping instance. By accepting this mapping the user would indicate a number of facts. Chiefly, that a match between <Number> field and <SKU#> field is not necessary to produce a valid output instance. This suggests that the composite foreign key between Source 1 and Source 2, is limited to (<Manufacturer>, <Name>). Conversely by rejecting this mapping the user would provide partial evidence that the field <SKU#> may be a necessary part of the composite foreign key between the two sources.

Central to our solution is a meta-model of all possible federating transformations. We syntactically decompose all possible federating view definitions and represent this universe using the Version Spaces model. The model is embodied in a tool, SPHINX, which interacts with users in an example driven format. SPHINX presents to the user sample data drawn from a data source as well as the record structure of the federated schema. (The user may also define a target schema on the fly.) In drag-and-drop fashion a user specifies a single record in the format of the federated schema. By virtue of having a meta-model of possible view definitions, the SPHINX system can detect if more than one federating view definition may materialize the example. If so, SPHINX generates and presents to the user an additional example. Rather than the user specifying additional semantic information or generating further examples, a user simply has to label the

proposed example as correct or incorrect. SPHINX exploits this feedback to prune its search space, and will keep submitting additional examples until the search converges towards a single, unambiguous federating view definition. SPHINX drives this active learning process using heuristics designed to minimize the number of examples a user is asked to inspect, and to maximize the information-gain value of each example. We further show how these heuristics may be refined using the same statistics used to parameterize query-cost models.

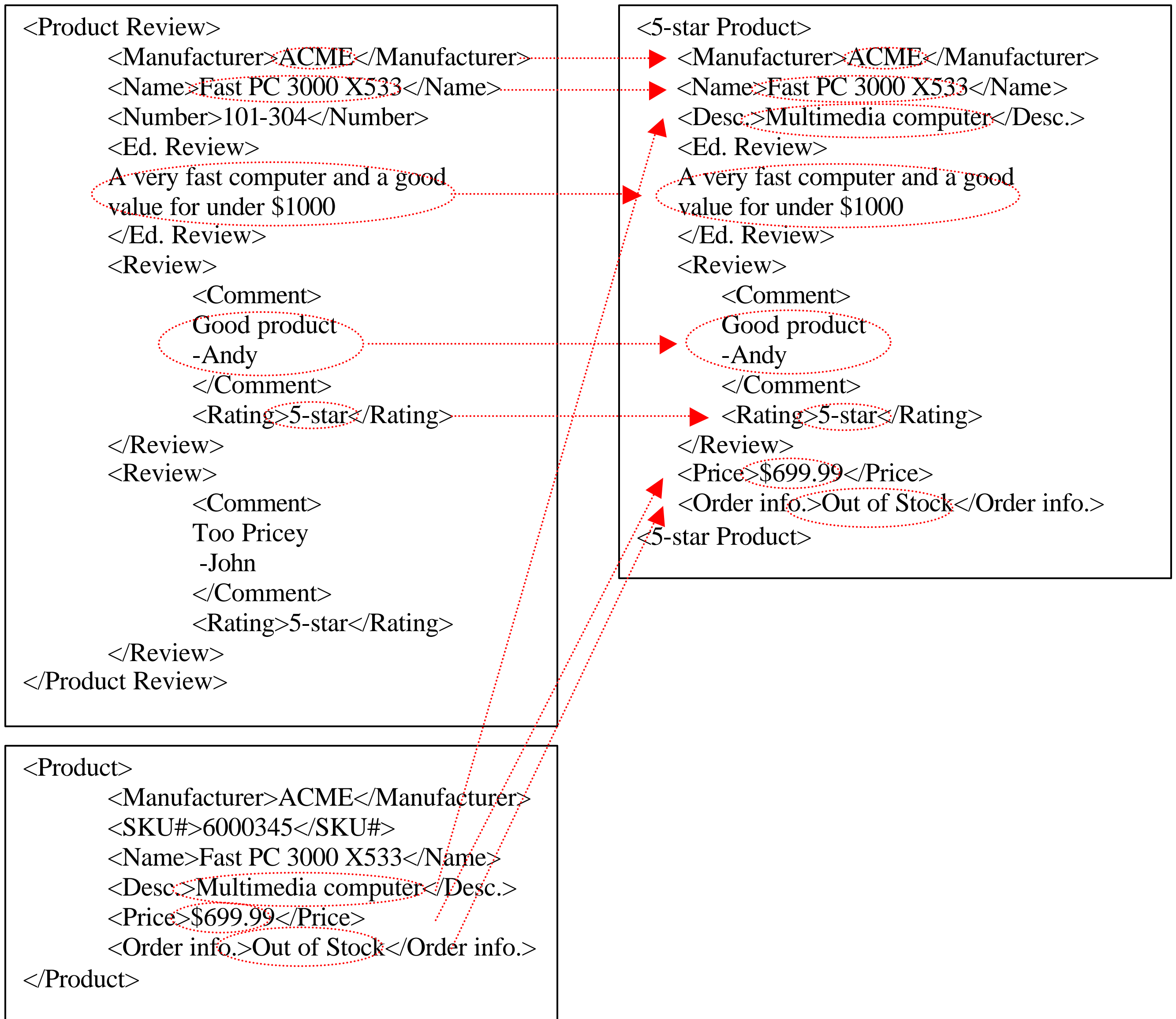


Figure 2 – Data Mapping Example

In Section 2, we review related work, and how our approach differs in nature from those of other existing works. In Section 3, we describe the learning problem and introduce a method to initialize for each particular integration problem, an appropriate Version Spaces model. We examine the expressive power of the class of queries defined by this meta-model. In Section 4 we define the SPHINX learning algorithm, which accepts data mapping examples and navigates the Version Spaces to converge to a solution. We prove in Section 5 that the algorithm is correct.

Namely it will converge towards the correct view definition, provided that the view definition was inside the scope of the initialized Version Spaces. Section 6 deals with the active learning portion of the SPHINX system, which generates and selects data examples to submit to the user. We describe one possible strategy to get Version Spaces to converge as quickly as possible towards the correct view definition. Accordingly, we introduce in Section 7 several heuristics, integrated in SPHINX, and motivated for the specific problem of integrating data from heterogeneous sources. We show with some experiments on various domains, that the number of examples required to drive the user interaction towards the correct view definition is linked to that view definition's size, and not to the size of the search space containing all potential view definitions. Finally in Section 8, we conclude and review future work and open problems.

2. Related Work

Mediator-based architectures to federate heterogeneous databases have drawn a lot of interest: see [1,6,10,11,16,27,30,31,32,34] to cite a few. In these systems, the basic assumption is that some highly qualified engineers may become domain experts and in one form or another write the specifications that will drive the data integration. In that line of work, several general-purpose languages for specifying heterogeneous data integration include SchemaSQL and SchemaLog [15], graph-based ontology [23] and XQuery [33]. As shown by Krishnamurthy, Litwin and Kent [14], such languages must possess higher order features to bridge schematic heterogeneities across data sources [10,13,34].

With the maturation of those systems, the problem of generating semantic specifications to federate data sources garnered more attention. Automated schema matching tools [4,5,17,18,21,23,24] were developed with the goal of helping an engineer cope with the plethora of domain information, which must be reconciled to produce a mapping between heterogeneous databases. Milo and Zohar [21] theorize that the vast majority of mappings between schema elements in heterogeneous databases are trivial. Those can be derived automatically, and user expertise can be saved for the truly complex mappings. Rahm and Bernstein [26] propose a taxonomy as part of a survey of automated schema matching tools. While often having high accuracy, these tools cannot guarantee that a correct mapping has been derived. A database specialist with domain expertise must examine the systems final output to verify the correctness. The average user cannot be expected to use the advanced query or semantic modeling languages commonly used to express those mappings.

Clio [20,35], and SPHINX are a little bit different from these systems. Both focus on the problem of query discovery for federated view definitions, and seek to combine both user interface and machine learning considerations. Clio contains a machine learning component which examines possible join paths in the query graph, and ranks possible view definitions, by estimating their likelihood. Clio then uses data examples to help the user decide between alternative mappings. In Clio, a user examines a set of illustrative examples as well as the output of the system for the top-ranked view definition, and renders judgment, whether it has converged on a correct transformation or whether additional examples must be included: "Clio helps the user understand the results of the mappings being formed and allows the user to verify that the transformations that result are what she intended" [35]. Clio does not embody a formal meta-model and does not generate the set of data examples which can uniquely fingerprint the correct federated view definition. With SPHINX, we now formalize the space of potential view definitions in such a way that it can interface with an active learning algorithm: this removes much of the burden of semantic verification and data browsing from the user. This model is based on the Version Spaces algorithm [12,22], which has been used in machine learning applications. To our knowledge, we are the first to exploit it in the context of databases.

3. Query Discovery

For the purpose of this exposition we will adopt a structured, hierarchical XML data model, where all XML documents conform to a DTD. Without loss of generality, we simplify this model by allowing only two types of nodes: element nodes and text nodes. There will be no optional elements. Only element nodes may have children and may be repeated. These restrictions do not affect the generality of our approach, since for any given XML DTD, an equivalent simplified DTD, can always be constructed by inserting additional element nodes to handle repetitions or optional nodes.

Source 1 document	{**Product Review**+}
Product Review	{Manufacturer, Name, Number, Ed. Review, **Review**+}
Review	{Comment, Rating}
Source 2 document	{**Product**+}
Product	{Manufacturer, SKU#, Name, Description, Price, Order info.}
Target Schema	{**5-star Product**+}
5-star Product	{Manufacturer, Name, Description, Ed. Review, **Review**+, Price, Order Info.}
Review	{Comment, Rating}

Figure 3 – Document descriptions

Figure 3 illustrates a DTD for the example in Figure 1, which will serve as the running example. Looking at the problem of query discovery to define views over heterogeneous data, we adopt the terms *target query* and *target view* as the goal concepts.

Definition: *Target View, Target Query*

The *target view* is the materialized view that the learning system (SPHINX) is trying to learn from the user. The *target view* is defined by the execution of the *target query* over the source databases.

The query discovery process is analogous to a guessing and elimination game between SPHINX and the user. The goal is for SPHINX to come up with a correct view definition, given a set of question/answer interactions with the user. It will naturally occur to the reader that over a given data source, there may be several queries, which yield the same materialized view as the target query. As such SPHINX cannot always converge on the correct target query. However, SPHINX will identify the entire class of queries, which correctly materialize the target view over the source data and picks the appropriate one (see Section 5- Lemma 5, Theorem1).

Figure 4 illustrates the overall question/answer approach to solving the query discovery problem. Note that this whole mechanism is hidden from the user, whose interaction is limited to answering yes or no.

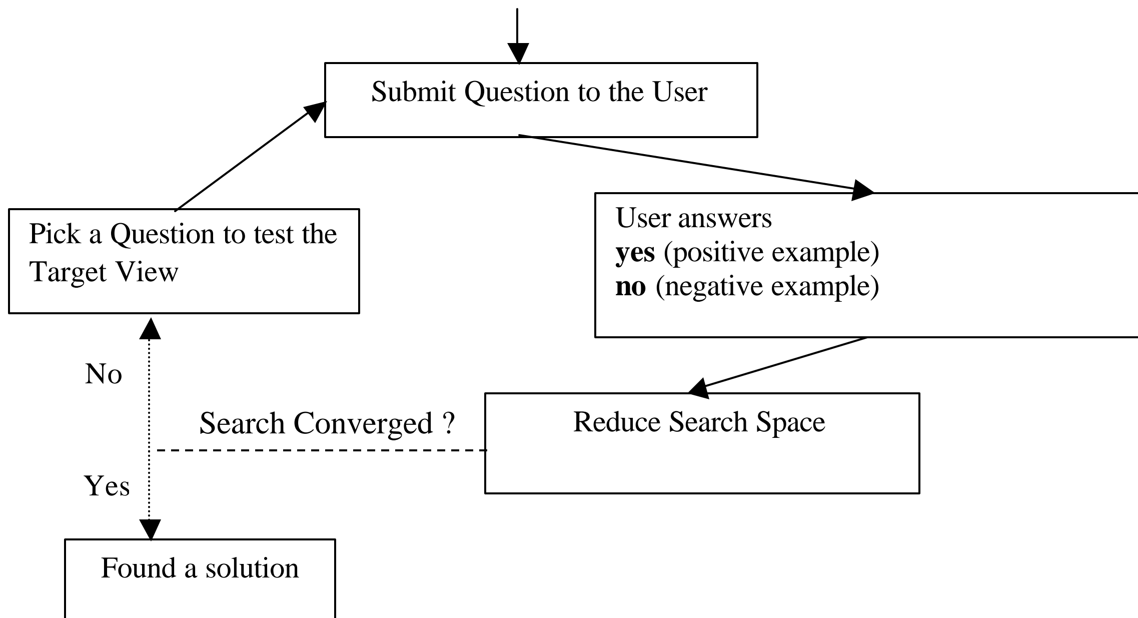


Figure 4 – Question/Answer Interaction

3.1 Query Discovery Framework

In this exposition we use XQuery as the language for view definition. However, the same query discovery process can take place in higher-order relational languages as well [2,15].

The first task is to delimit the search space. We ask the user to supply the first data-mapping example, such as the one shown in Figure 2. The user, with a point-and-click browsing tool, select values and drags them to an existing target schema or one of his own definition.

```
For     $a in document('source1')/product review,
        $b in document('source 1')/product review/review,
        $c in document('source2')/product
Where   value-equals($a/reviews, $b) and
        $a/manufacturer = $c/manufacturer and
        $a/name = $c/name and
        $b/rating = '5-star'
Return  <5-star Product>    <manufacturer>$a/manufacturer</manufacturer>
                            <description>$b/description</description>
                            <ed. review>$a/ed. review</ed. review>
                            <reviews><comment>$b/comment</comment>
                                  <rating>$b/rating</rating></reviews>
                            <price>$b/price</price>
                            <order info.>$b/order info.</order info.>
        </5-star Product>
```

Figure 5 – XML View Definition

Consider the XQuery view definition in Figure 5. It illustrates a possible view definition for Figure 2. Most of that view definition can be derived syntactically from the source schema and the data mapping shown in Figure 2. For each internal node in the source schemas: <product

review>, <reviews> and <product>, a variable is introduced in the *For* clause: respectively \$a, \$b and \$c. The *Let* clause is empty. The *Where* clause contains a list of predicates. Those predicates verify: that \$a and \$b are properly nested, that (<manufacturer>,<name>) forms a composite foreign key between \$a and \$c, and that only products rated '5-star' will appear in the defined view. Note that that the query shown in Figure 5 outputs an un-nested structure. Nesting the structure according to the specifications of the target schema is not hard and does not need to be addressed here.

Consider the following relational formula encapsulating the kind of query, which can be expressed with the syntactic techniques illustrated in Figure 5:

$$Q(E_1, ..., E_k, P_1, ..., P_m, O_1, ..., O_n) = \Pi_{E1, ..., Ek} (\sigma_{P1 \wedge P2 \wedge ... \wedge Pm} (O_1 \times O_2 \times ... \times O_n))$$

Formula 1 – Standard View Definition Formula.

Formula 1 fully defines a query once the following parameters are defined: O_1, O_2, ...O_n are object sets, and implicitly, $\$r_1$, $\$r_2$, ... , $\$r_n$ are the variables ranging over those sets. P_1, P_2, ...P_m are Boolean predicates formed using variables $\$r_1$, … , $\$r_n$. Expressions E_1, E_2, ..., E_k are projection expressions also formed using variables $\$r_1$, … , $\$r_n$. Together O_1, O_2,...,O_n, P_1, ...P_m, E_1,...E_k define the query Q (E_1, ..., E_k,P_1, ..., P_m, O_1, ..., O_n).

The query shown in Figure 5 can be expressed within the framework of Formula 1. A large class of queries Project-Select-Join can be expressed in this framework. Section 3.4 will examine the impact of Formula 1 on the generality of our query discovery.

3.2 Building the Version Spaces model.

We build a Version Spaces model, which we will use to describe and keep track of all queries in a delimited search space within the framework of Formula 1. To that end, we will execute the following tasks:

- The first task is to identify a number of Cartesian sets O_1, …, O_n. These represent the object sets which will appear in the *For* clause of the target query (Algorithm-1).

- The second task is to construct the projection expressions E_1,...,E_k which will be used in the *Return* clause (Algorithm-2).

- The third task is to assemble the potential set of predicates P_1, …, P_{pf} which can be built over O_1, …, O_n and could be used in the *Where* clause of the target query (Algorithm-3)

These three steps identify what we can learn about the target query by exploiting the initial data-mapping example. After performing these steps, the *For* clause and *Return* clause for the target query will be determined by Algorithm-1 and Algorithm-2. The only remaining work will be to identify the proper subset of filter predicates $\{P_1, …, P_m\}$ of $\{P_1, …, P_{pf}\}$ which appear in the *Where* clause. Thus the search space for the target query is exactly the powerset of the set $\{P_1, …, P_{pf}\}$. The Version Spaces model feasibly embodies the search space as a collection of Boolean vectors of size pf over the possible predicates. Where unsupervised learning approaches are initially faced with a seemingly infinite space of target views, a single insightful example provided by the user both structures the search, and, as supported by our empirical results immediately reduces the search space to a manageable size (2^{pf}).

3.2.1 User Supplied Data Mapping Example

A data-mapping example supplied by the user is initially encoded in a sequence of the type: ($Node_1$, $Node_2$, …, $Node_r$). $Node_i$ represents a node in the source documents, which is

mapped to the ith position of the output. Let us illustrate this construction with an example. Consider the mapping shown in Figure 2. There are 8 source nodes mapped to the output:

$Node_1$ = document('source1')/product review1[*]/manufacturer
$Node_2$ = document('source1')/product review1/name
$Node_3$ = document('source1')/product review1/ed. review
$Node_4$ = document('source1')/product review1/review1/comment
$Node_5$ = document('source1')/product review1/review1/rating
$Node_6$ = document('source2')/product1/desc.
$Node_7$ = document('source2')/product1/price
$Node_8$ = document('source2')/product1/order info.

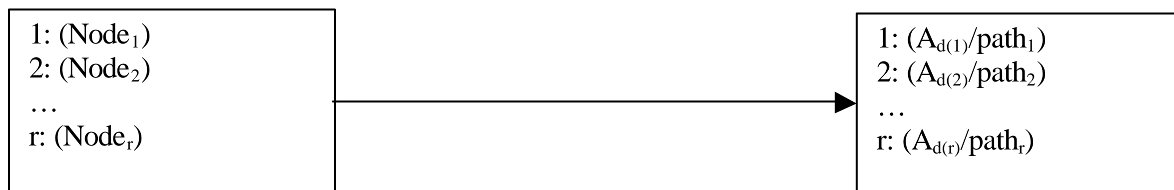
We look at all the nodes and identify the set of ancestor nodes $\{A_1, A_2, \dots A_n\}$ with the following properties:

- A_i is an internal node in a source document
- A_i is either the ancestor or equal to some mapped source node $Node_j$, 1 j r

In our example, this construction yields the following set of nodes:

A_1: document('source1')/product review1
A_2: document('source1')/product review1/review1
A_3: document('source2')/product1

This set of ancestor nodes is used to rewrite nodes $Node_1$, $Node_2$, … , $Node_r$ in the data-mapping example. As in our example:
$Node_1$ = A_1/manufacturer
$Node_2$ = A_1/name
$Node_3$ = A_1/ed. review
$Node_4$ = A_2/comment
$Node_5$ = A_2/rating
$Node_6$ = A_3/desc.
$Node_7$ = A_3/price
$Node_8$ = A_3/order info.



3.2.2 Cartesian Object Sets

Every variable defined in the *For* clause ranges over an object set of internal nodes in the source documents. Before application of join predicates, these object sets participate in a Cartesian product, as shown in Formula 1. We introduce such an object set for each node in the set $\{A_1, \dots, A_n\}$, and assign to each a new variable symbol $\$r_i$. Each object set is uniquely identified by a path expression over the source documents.

[*] Since node elements can be repeated, our numbering system will distinguish an individual node from its siblings.

Algorithm-1. Generate *For* clause

- for every node A_i, in $\{A_1, \ldots, A_n\}$
  such that its source path is '$x_1/x_2/../x_p$' in source s
  insert "$\$r_i$ in document('source s')/$x_1/x_2/../x_p$" in the *For* clause.

Algorithm-1 applied to the Figure 2 example yields the following *For* clause:

"For $\$r_1$ in document('source1')/product review,
$\$r_2$ in document('source1')/product review/review,
$\$r_3$ in document('source2')/product "

3.2.3 Projection Expressions

For our purposes, we model the *Return* clause as a template, which exclusively uses the projection expressions $E_1, \ldots, E_k$ from Formula 1 as fillers.

Algorithm-2. Generate Projection Expression

- for every mapped source node ($A_{d(i)}$/$path_i$)
  create projection expression E_i = "$\$r_{d(i)}$/$path_i$)

Algorithm-2 applied to the Figure 2 example creates the following projection expressions:

E_1 = $\$r_1$/manufacturer
E_2 = $\$r_1$/name
E_3 = $\$r_1$/ed.review
E_4 = $\$r_2$/comment
E_5 = $\$r_2$/rating
E_6 = $\$r_3$/desc.
E_7 = $\$r_3$/price
E_8 = $\$r_3$/order info.

3.2.4 Potential Features

A *potential feature* is a filtering predicate in the *Where* clause, which is defined on the object variables introduced in the *For* clause. Figure 5, shows a query with two kinds of potential features actually appearing in its *Where* clause: a foreign key constraint, and a match between an element and a specific value. Both kinds are equality predicates: they are formed using only variables defined in the *For* clause, arbitrary constants and an equality operator ('=' for simple types, 'value-equals' for objects)The number of potential features involving equality predicates is finite and can be derived automatically from the instantiated values of the initial data-mapping example. In other words, we can reconstitute the exact set of equality predicates, which may legally appear in the target query. That set is guaranteed to be complete because we observe that

any equality predicate not within that set could not appear in the *Where* clause target query, since it would be negated by, and hence incompatible with the user supplied example.

Algorithm-3 proposes the generation of almost all those equality predicates. However, because variables are introduced in the *For* clause only for those object sets which appear in the mapping or for their ancestors, Algorithm-3 does not generate all legal equality predicates.

Algorithm-3 applied to the Figure 2 example yields the following set of potential features:

P_1: $\$r_1$ = “<Manufacturer>ACME… …”
P_2: $\$r_2$ = “<Comment>Good Product… …”
P_3: $\$r_3$ = “<Manufacturer>ACME… …”
P_4: $\$r_1$/manufacturer = “ACME”
P_5: $\$r_1$/name = “Fast PC 3000 X533”
P_6: $\$r_1$/ed. review = “101-304”
P_7: $\$r_2$/comment = “Good Product… …”
P_8: $\$r_2$/rating = “5-star”
P_9: $\$r_3$/manufacturer = “ACME”
P_{10}: $\$r_3$/SKU# = 6000345
P_{11}: $\$r_3$/name = “Fast PC 3000 X533”
P_{12}: $\$r_3$/desc. = “Multimedia computer”
P_{13}: $\$r_3$/price = “\$699.99”
P_{14}: $\$r_3$/order info.=”Out of Stock”
P_{15}: value-equals($\$r_1$/review, $\$r_2$)
P_{16}: $\$r_1$/manufacturer = $\$r_3$/manufacturer
P_{17}: $\$r_1$/name = $\$r_3$/name

${}_i$/$name_i$ = val_i”, where $name_i$ and val_i are respectively the schema name and value of the child node.

- for every pair A_i , A_j, associated respectively with values val_i and val_j and with variables $\$r_i$ and $\$r_j$, such that val_i=val_j, create the potential feature: “$\$r_i = \$r_j$”
- for every pair A_i , A_j, associated respectively with variables $\$r_i$ and $\$r_j$, such that A_i is the direct parent of A_j, create the potential feature: “value-equals ($\$r_i$/$name_j$, $\$r_j$)”, where $name_j$ is the schema name of A_j
- for every pair A_i , A_j, associated respectively with variables $\$r_i$ and $\$r_j$, such that A_i has a non-leaf child ($name_i$, val_i), and such that A_j has value val_j, and such that val_i = val_j, create the potential feature “$\$r_i$/$name_i$ = $\$r_j$”
- for every pair A_i , A_j, associated respectively with variables $\$r_i$ and $\$r_j$, such that they each respectively have leaf-node children ($name_i$, val_i) and ($name_j$, val_j) and such that val_i=val_j, create the potential feature: “$\$r_i$/$name_i$ = $\$r_j$/$name_j$”

3.3 Boolean Feature Vector

In the rest of this discussion we will assume that the set of potential filter predicates, or features: $\{P_1, P_2, \ldots, P_{pf}\}$ has been assembled, either by applying Algorithm-3, or by other means. We introduce the notion of *feature vector* as a Boolean vector of size pf. A *query feature vector* is the practical and unambiguous specification of a query in our model.

Definition: *Query Feature Vector*

The *query feature vector* associated with a query q, abbreviated FV(q) is a Boolean vector of size pf. Given the initial data-mapping example, if $FV(q) = (q_1, \ldots, q_{pf})$, q is the query with a *For* clause generated by Algorithm-1, an empty *Let* clause, a *Return* clause generated by Algorithm-2, and such that $q_i=1$ if and only if filter predicate P_i appears in the *Where* clause of q.

There are exactly 2^{pf} queries in the Search space, 2^{pf} distinct feature vectors, and we have defined a one-to-one mapping between queries and their query feature vectors.

Definition: *more specific than, more general than*

Let a and b be two feature vectors such that $a = (a_1, \ldots, a_{pf})$ and $b = (b_1, \ldots, b_{pf})$:

- a is *more specific than* b (noted $a \geq b$) if and only if ($\forall i$: $a_i \geq b_i$)
- b is *more general than* a if and only if a is *more specific than* b.

3.4 Expressive Power of our Approach

To place this work into a larger context we review first what is currently within the scope of our Version Spaces model, and second of our feasibility prototype, SPHINX. We speak to the scope of features and predicates that can be addressed in this model and those that must be addressed in other ways.

Our data model is relevant for structured XML documents and object relational databases. This is equivalent to stating SPHINX handles mappings of XML data structured in conformity with a DTD. The SPHINX approach is not appropriate for defining arbitrary tree mappings, or handling semi-structured data. Skolem functions to generate new object identifiers are absent, which prohibits the construction of DAGs with SPHINX [9,19,28]. Instead SPHINX focuses on generating higher order SchemaSQL or XQuery view definitions, which are based on the Project-Select-Join model of Formula-1.

The Version Spaces model can handle any query, formed with a conjunction of arbitrary predicates of any kind. As such predicates may contain any arithmetic such as 'x.a+3>y.a', or more complex features such as negation 'x.a≠y.a', outer joins '(x.a=y.a or x.a=null or y.a=null)', disjunction, etc.. Such queries are expressible with Formula-1 and fall within the expressive power of Version Spaces and of the active learning algorithm for which we elaborated formal correctness and termination guarantees.

In our feasibility prototype, we incorporated Algorithm-1, 2 and 3, as a fully automated way of building a search space with a limited class of queries (equality predicates) and with no user intervention. With this initial GUI prototype, we explore how far a fully automated system with almost no user interaction can go, in an area that has traditionally challenged our ability to propose GUIs suited for non-technical users. Later, a more advanced GUI prototype will give knowledgeable users the opportunity to enter additional input in simple pull-down menus.

Further, as our goals were to push the envelope on a fully automated system, we do not handle unit conversions, or aggregation. However conversions such as ounces to grams, months

to quarters, and other sorts of data misrepresentations can readily be identified by a user and handled by a simple and limited GUI. However, the construction of whole synonym dictionaries (e.g. 'Pentium 3' vs. 'Pentium III') or thesauri is characterized by a radically different set of technical challenges [25,29] and is well outside the scope of our study.

In choosing Algorithm-1, 2 and 3 for our prototype implementation of SPHINX, we explicitly curtailed the completeness of the search space to a limited class of queries with equality predicates. Without additional input from the user, currently generated potential predicates include all legal equality selection predicates on objects involved in the initial data mapping. They also include all legal equality join predicates on those same objects with the exception of self-joins and of a peculiar kind of join, characterized by join paths in the query graph that traverse children of data mapping elements. As shown in [20], a more exhaustive generation of all equality selection and join predicates is possible, and would require building a query graph and introducing in the *For* clause, every possible join path from the data mapping instance.

4. Example-Based Learning with the Version Spaces Model

The Versions Spaces algorithm was first presented by Mitchell [22]. For our application we've extended that algorithm to include the concept of active-learning: the system suggests additional examples rather than seeking them directly from the user, whose role is limited to deciding if the examples presented to him form a valid row in the target view.

The SPHINX learning algorithm performs an iterative loop, whose termination condition is the convergence of the Version Spaces. At each step a feature vector is chosen, and labeled with one of three labels: *positive*, *negative* or *missing*. As a result of these repeated steps the target query can be narrowed down to a dwindling subset of the search space by the application of three rules. This narrowing subset is commonly called the *space of remaining hypothesis*. The Version Spaces model inside the learning algorithm tracks the space of remaining hypothesis, and converges when that space is reduced to a single query.

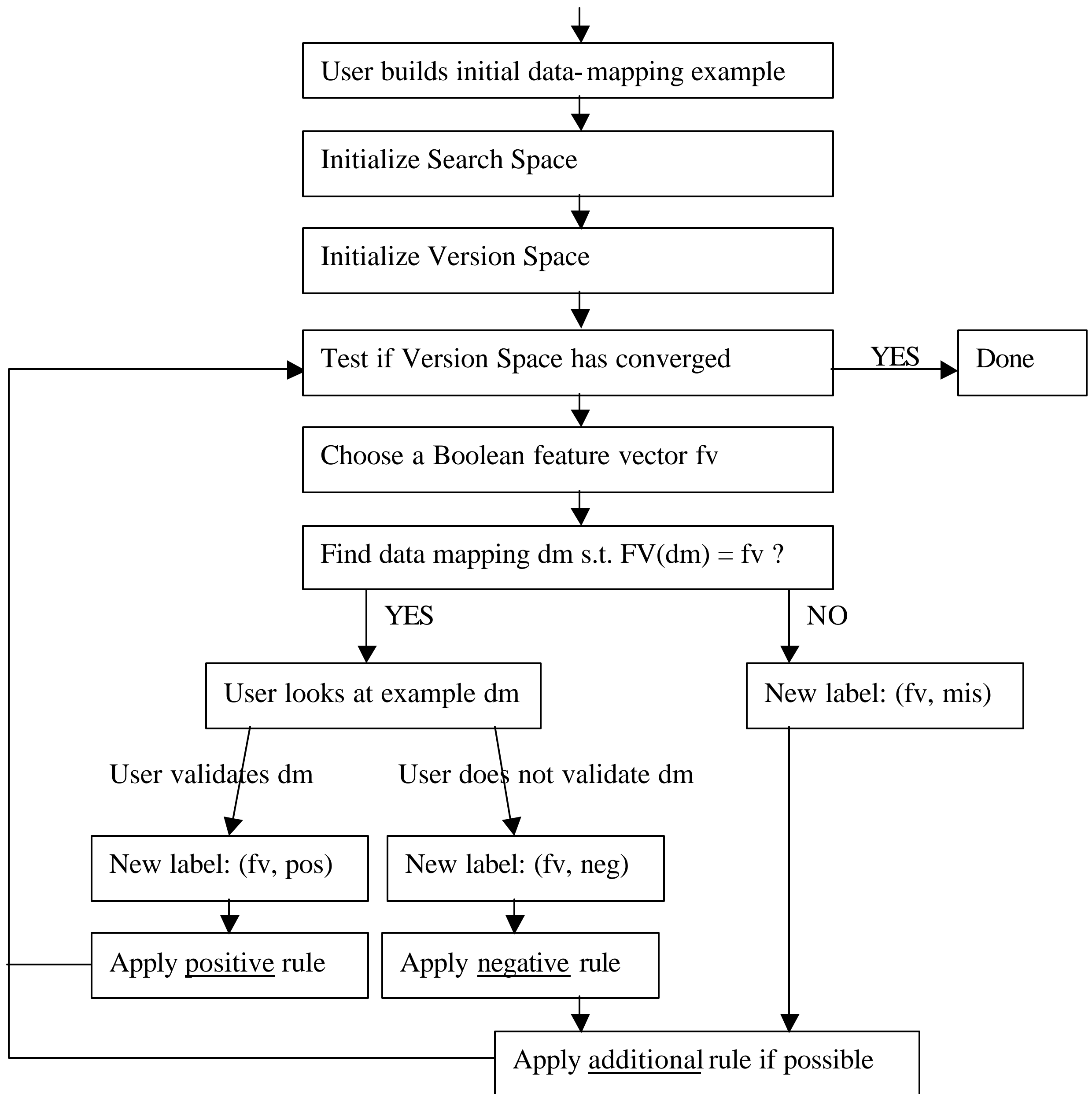


Figure 6 – Chain of Events in the SPHINX active learning algorithm

4.1 Version Spaces State

We introduce the concept of Version Spaces state for the purpose of tracking the space of remaining hypothesis.

Definition: *Version Spaces state*
A Version Spaces state is a pair of items **(s, G)** such that:

- s is a query feature vector called the most specific feature vector
- G is a set of query feature vectors called the most general set.

As illustrated in Figure 7, the Version Spaces is initialized with the initial state (s_0, G_0), with $s_0 = (1, 1, \ldots, 1)$ and $G_0 = \{ (0, 0, \ldots, 0) \}$.

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Most Specific Feature Vector

{

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

}

Most General Feature Vector Set

Figure 7 – Initial Version Spaces state

We define the notion of *query set* for a given Version Spaces state as the space of remaining hypothesis. The initial query set QS(s_0, G_0) is equivalent to the entire search space.

Definition: *Query set*

Let (**s**, **G**) be a pair of items such that the first item **s**, is a query feature vector and the second item **G** is a set of feature vectors: **G** = $\{g_0, g_1, \ldots, g_{mg}\}$.

Query set QS(**s**, **G**) = {q | q is more general than s and $\exists g_j \in G$ such that g_j is more general than q}.

4.2 Data Mapping

We defined the notion of data mapping instance to formalize the concept of source data coming together to form an object in the target schema (as previously illustrated in Figure 2). A data mapping is a positive example if the given source data correctly forms a member of the target view.

Definition: *Data Mapping*

A *data mapping* instance dm is an assignment (o_1, o_2, …, o_n) of variables $\$r_1, \$r_2, \ldots, \$r_n$ in their respective object sets O_1, O_2, …, O_n:

$dm = (o_1, o_2, .. , o_n) \in O_1 \times O_2 \times .. \times O_n$

Definition: *positive, negative data mapping example*

A *data mapping* instance dm = (o_1, o_2, …, o_n) is said to form a *positive example* if and only if the target schema object formed by $Đ_{E1, ..., Ek}$ (o_1, o_2, … , o_n) is a member of the target view.

If not, dm is said to form a *negative example*.

Figures 2, 8 ,10, 14 show the graphical representation for a data mapping. A data mapping can always be represented by showing a set of the data values in the source databases, combining to form an element in the target schema. If the element produced is a member of the target view, the data mapping represents a positive example (Figures 2, 8, 14), if not it represents a negative example (Figure 10).

Just as we did for queries, we can associate a Boolean feature vector with each data mapping instance dm.

Definition: *Example Feature Vector*
For a given data mapping dm, the *example feature vector* FV(dm) is defined as
$FV(dm) = (P_1(\$r_1 = o_1, \ldots, \$r_n = o_n), P_2(\$r_1 = o_1, \ldots, \$r_n = o_n), \ldots, P_{pf}(\$r_1 = o_1, \ldots, \$r_n = o_n))$.

4.3 Rule Definition

We introduce the concept of rules, and the three kinds of rules used in the SPHINX learning algorithm. Formally we define a rule as one of three operators acting on a Version Spaces state. In turn, we will formally define three rules.

Definition: *rule*
A rule is an operator, which takes a Version Spaces state (s, G) and a feature vector fv as input and returns a new Version Spaces state (s', G'). Applying a rule r moves a Version Spaces in state S with vector, to a new state S' with vector

4.3.1 Positive rule

The positive rule operator R_p modifies the Version Spaces state by eliminating from the query set those queries that are inconsistent with a given positive data mapping.

Definition: *Positive rule operator*
Let dm be a data mapping such that $FV(dm) = (e_1, \ldots, e_{pf})$. The positive rule operator $R_p((e_1, \ldots, e_{pf}))$ operates on a Version Spaces state (s, G), $s = (s_1, \ldots, s_{pf})$, $G = \{g_1, .., g_n\}$, and $g_i = (g_{i,1}, g_{i,2}, \ldots, g_{i,pf})$.
$\mathbf{R_p((e_1, \ldots, e_{pf}))}$**:** $(s,G) \rightarrow (s',G')$

- $s' = (s_1', \ldots, s_{pf}')$ such that
  - $e_i = 1 \Rightarrow s_i' = s_i$
  - $e_i = 0 \Rightarrow s_i' = 0$
- $G' = G$

Consider the positive data mapping instance dm_1 shown in Figure 8. The mapped values are circled. Because the data in dm_1 is substantially different from the initial data-mapping example in Figure 2, a large number of the potential features predicates do not hold on dm_1: P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_9, P_{10}, P_{11}, P_{12}, P_{13}, P_{14}. Conversely the following predicates do hold on dm_1: P_8, P_{15}, P_{16}, P_{17}. Thus the example feature vector for dm_1 is: $FV(dm_1) = (0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1)$. The result of the subsequent application of $R_p(FV(dm_1))$ to the initial Version Spaces state (Figure 7) is shown in Figure 9. Only the most specific vector is modified: all potential features which are negated in dm_1 see their vector value lowered from 1 to 0. Potential features whose predicates are still fulfilled (P_8, P_{15}, P_{16}, P_{17}) suffer no change.

The changes shown in Figure 9 eliminate from the query set, all the queries with any of the negated predicates (P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_9, P_{10}, P_{11}, P_{12}, P_{13}, P_{14}) in their *Where* clause: since those queries would not produce a target view where dm_1 could be a positive example. This property is formally stated in Lemma 1.

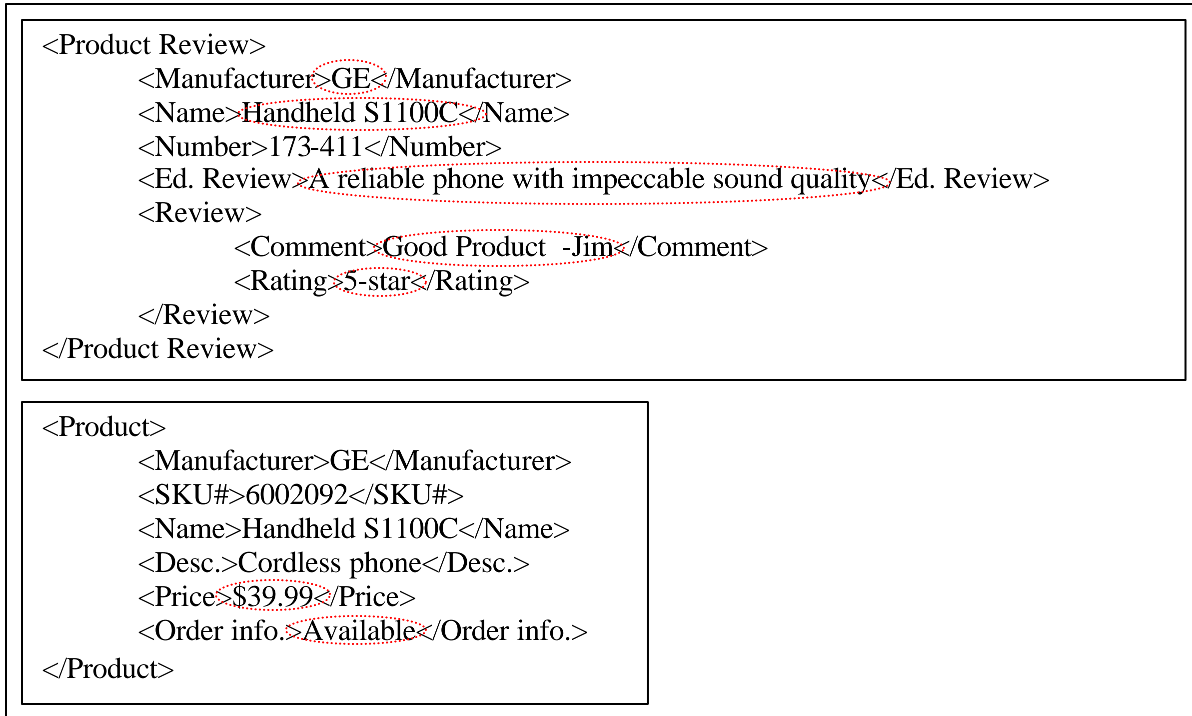
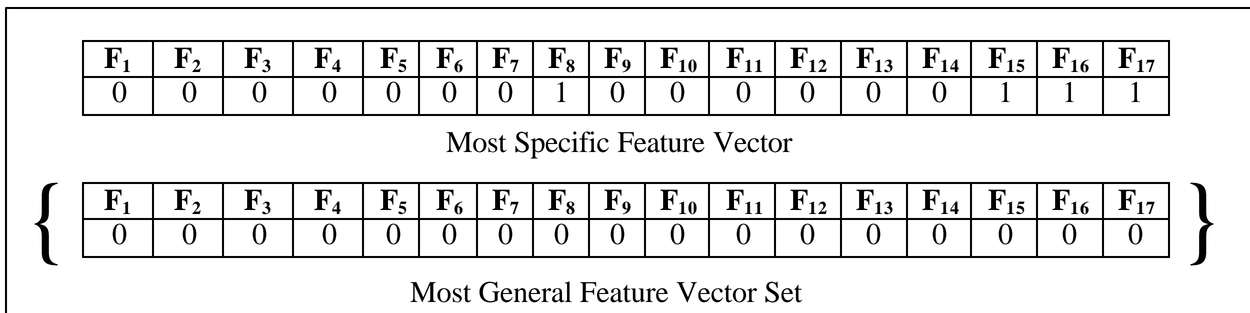


Figure 8 – Positive Data Mapping Example dm_1



F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}
0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	1

Most Specific Feature Vector

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Most General Feature Vector Set

Figure 9 – Applying the *positive rule* operator $R_p(FV(dm_1))$ to the initial Version Spaces state.

4.3.2 Negative rule

The negative rule operator R_n is the more complex of the three and modifies the Version Spaces state in order to eliminates from the query set those queries that are inconsistent with a given negative data mapping.

Definition: *Negative rule operator*

Let dm be a data mapping, such that $FV(dm) = (e_1, \ldots, e_{pf})$.

The negative rule operator $R_n((e_1, \ldots, e_{pf}))$ operates on a Version Spaces state (s, G), with $s = (s_1, \ldots, s_{pf})$, $G = \{g_1, .., g_n\}$, and $g_i = (g_{i,1}, g_{i,2}, \ldots, g_{i,pf})$.

$R_n((e_1, \ldots, e_{pf}))$: $(s,G) \rightarrow (s'', G'')$

- $s'' = s$
- $G' = \{ (g_{i,1}', \ldots, g_{i,pf}') \mid [(\exists p: (g_{p,1}, \ldots, g_{p,pf}) \in G) \wedge (\forall j: e_j=0 \Rightarrow g_{p,j} = 0)) \wedge (\exists f: (\forall j \neq f: g_{i,j}' = g_{p,j}) \wedge e_f=0 \wedge g_{i,f}' = 1)]$
  $\vee$ $[(\exists p: (g_{p,1}, \ldots, g_{p,pf}) \in G) \wedge (\exists f: e_f=0 \wedge g_{p,f} = 1) \wedge (\forall j: g_{p,j} = g_{i,j}')] \}$
- $G'' = \{ (g_{i,1}', \ldots, g_{i,pf}') \mid (g_{i,1}', \ldots, g_{i,pf}') \in G' \wedge (g_{i,1}', \ldots, g_{i,pf}') \leq s' \}$

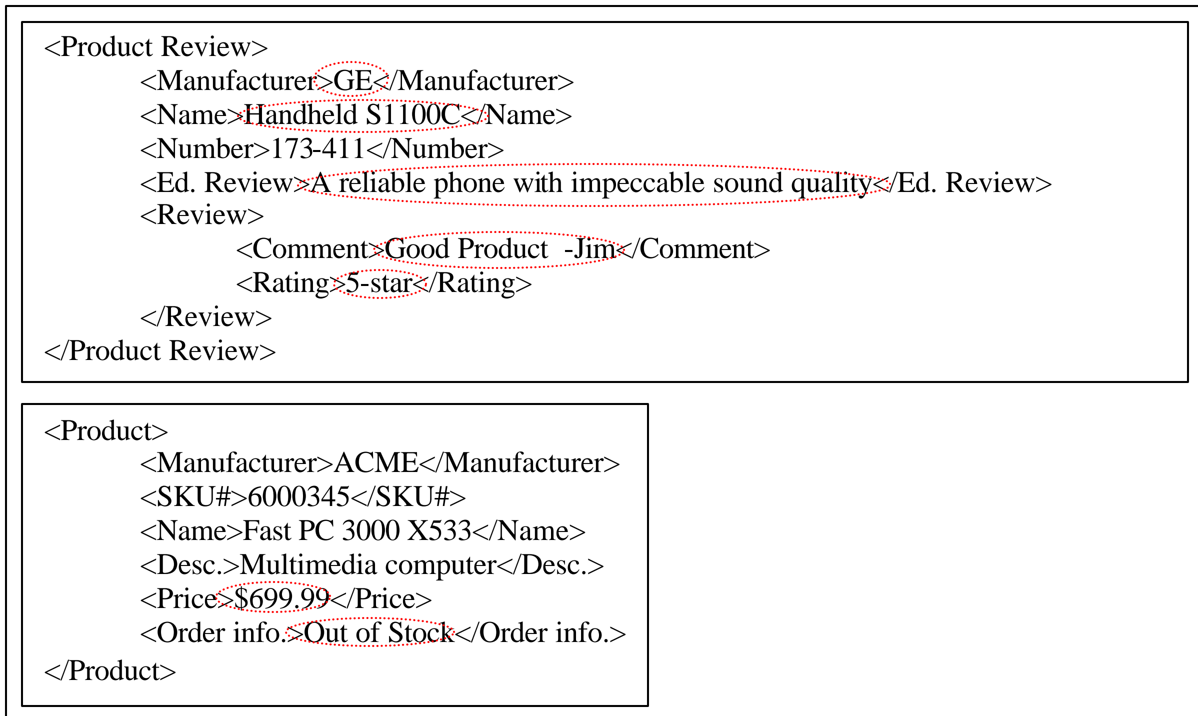


Figure 10 – Negative Data Mapping Example dm_2

Consider the data mapping dm_2 shown in Figure 10. It differs slightly from dm_1, in that the data for source 1 is the same (the product is a GE handheld S1100C), but it is matched with a totally different product from source 2. The feature vector for dm_2 is $FV(dm_2) = (0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0)$. There are 7 potential features whose predicates are negated in dm_2: $(F_1, F_2, F_4, F_5, F_6, F_{16}, F_{17})$.

The application of the $R_n(FV(dm_2))$ operator leaves s, the most specific vector unchanged and applies only to G, which has only one member at this stage: $G = \{g_0\}$. The application of $R_n(FV(dm_2))$ happens in two phases:

- In the first phase a new vector g_i' is created from g_0 by changing the bit of exactly one of the 7 features to 1. Since there are 7 features there will be 7 new vectors: $g_1' = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)$, $g_2' = (0, 1, 0, 0\ldots, 0)$ up to $g_7' = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)$.
- In the second phase only those vectors which are still more general than s will be kept. This eliminates all but g_6' and g_7', thus $g_1'' = g_6'$ and $g_2'' = g_7'$. Thus the final result is shown in Figure 11.

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}
0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	1

Most Specific Feature Vector

{

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

}

Most General Feature Vector Set

Figure 11 – Applying the *negative rule operator* $R_n(FV(dm_2))$ to the Version Spaces state

4.3.3 Additional rule

The additional convergence rule operator $R_a(p)$ is applied in the negative and missing label branches every time its precondition is met for some potential feature p, 1 p pf. If the precondition is not met for any p, the Version Spaces state is unchanged and nothing further happens. If it is met for some p, then the operator $R_a(p)$ is applied and the Version Spaces state is modified to reflect the information derived from the precondition being fulfilled.

Definition: *Set* 0_p
The set 0_p is the set of feature vectors containing a zero at the pth position.
$0_p = \{ (e_1, e_2, \ldots, e_{pf}) \mid e_p = 0\}$

Precondition $R_a(p)$:
Precondition(p) is met, if for any data mapping dm, FV(dm) $\in 0_p$ implies that dm is a negative example. If there is no data mapping dm, such that FV(dm) $\in 0_p$, then Precondition(p) is true.
($\forall$dm : FV(dm) $\in 0_p \Rightarrow$ dm is a negative example) $\Rightarrow$ Precondition(p) is true

Definition: *Additional rule operator (Applied when Precondition(p) is true)*
$R_a(p)$: (s, G) $\rightarrow$ (s', G')

- G' = G
- $s' = (s_1', \ldots, s_{pf}')$, $s_p' = 1 \wedge (\forall i \neq p : s_i' = s_i)$

The original Version Spaces algorithm did not require such a rule. The necessity for the *additional rule* is dictated by the demands of a sample selection system. There will be cases when no data mapping instance for a given feature vector can be found from the existing data sources. Consider predicate P_8: $\$r_2$/rating = "5-star". Assume all objects in the source have the rating "5-star". Thus all positive data mapping instances validated by the user must contain the rating "5-star". This is insufficient to prove that predicate P_8 is part of the target view since all negative data mappings must also contain the "5-star" rating. This makes disproving P_8 impossible. Thus, since no examples can be found with a different rating, the system will never be able to determine if the predicate P_8 should appear or not in the *Where* clause of the target concept.

It should be noted that since all data instances in the source fulfill predicate P_8, its presence in the target query does affect the content of the target view. Every query q containing

P_8 in the *Where* clause, has an identical counterpart q' which is similar to q except q' is missing P_8 in the *Where* clause. It is always the case that q and q' materialize the same target view. In that case, applying the *additional rule operator $R_a(8)$* will reduce the query set by removing for every query q, its counterpart q'.

4.3.4 Impact on Updates

Consider the scenario where the Additional rule operator $R_a(8)$ is applied because no data mapping can be found with a rating other than "5-star". Assume that as the result of an update or a modification of the source data, a rating of "4-star" appears at some point in the future. It will be necessary for SPHINX to re-evaluate the target concept against this new information.

Thus every application of the *additional rule operator* should be logged, and a trigger should be introduced as an integrity constraint on the source data.

For each application of an additional rule operator $R_a(p)$, Precondition(p) must become an integrity constraint on the source data. Such an integrity constraint is violated when a data mapping dm appears in the source such that:

- $FV(dm) \in 0_p$
- dm has not been labeled a negative data mapping.

Violation of this integrity constraint, and of Precondition(p) must trigger a restart of the learning algorithm at the point where the additional rule operator $R_a(p)$ was applied.

5. Correctness

Per Section 3.4 we discussed the relative completeness of the search space with respect to a limited class of view defining queries. In this section we look at correctness and prove that the SPHINX learning algorithm, if it converges, correctly converges to the correct target concept in the search space. To do so, we prove that each rule operator application removes from the space of remaining hypothesis only those queries, which are inconsistent with the data mapping labels. We also prove that the learning algorithm terminates after a finite number of steps.

5.1 Data Mapping and View Mapping Set

The Data Mapping set corresponds to the Cartesian product of the Cartesian sets O_1, O_2, …, O_n identified by Algorithm-3. We also define the positive data mapping set as a subset.

Definition: *Data Mapping Set DM*
The set DM of data mapping instances is formally defined as the Cartesian product of the object sets O_1, O_2, … O_n.

$$DM = O_1 \times O_2 \times \ldots \times O_n$$

Formula 2 – Data Mapping set

Definition: *Positive Data Mapping Set DM^+*
Given a query q, such that $FV(q) = (q_1, \ldots, q_{pf})$, the positive data mapping set $DM^+(q)$ is the subset of DM which does not negate any of the predicates included in the query q.

Formula 3 – Positive Data Mapping set defined by a query

$$DM^+(q) = \{ (o_1, \ldots, o_n) \in DM \mid \forall i : q_i \Rightarrow P_i(\$r_1 = o_1, \ldots, \$r_n = o_n) \}$$

Note that if q_0 is the query with feature vector $FV(q_0) = (0, 0, \ldots, 0)$, then $DM^+(q_0) = DM$
Note that if a and b are two queries such that a is more specific than b then $DM^+(a) \subseteq DM^+(b)$.

5.2 View Defined by a Query

Any query q in the search space defines a view over the set of source databases. Two different queries q_1 and q_2 may define the same view. A trivial illustration of this is when the source databases are empty.

Definition: *View Defined by a Query q*
Given a query q, such that $FV(q) = (q_1, \ldots, q_{pf})$, we note View(q) the view defined by q.

$$View(q) = \{ \prod(E_1(\$r_1=o_1, \ldots, \$r_n=o_n), \ldots, E_k(\$r_1=o_1, \ldots, \$r_n=o_n)) \mid (o_1, \ldots, o_n) \in DM^+(q)$$

Formula 4 – View defined by a query

Formula 5 briefly illustrates for a query q the relationship between DM, $DM^+(q)$ and View(q).

$$View(q) = Ð_{(E1, \ldots, Ek)} (DM^+(q)) = Ð_{(E1, \ldots, Ek)} (\sigma_{P1, \ldots Pm} (DM))$$

Formula 5 – Relationship between View, DM^+ and DM.

In a machine learning view of the query discovery problem for target query q_t, elements of DM serve as labeled examples: positive examples belong to $DM^+(q_t)$, and negative examples do not belong to $DM^+(q_t)$.

Lemma 1 states that a data mapping dm is a positive example for query q if and only if feature vector FV(dm) is more specific than feature vector FV(q).

Lemma 1:
Let dm be a data mapping such that $FV(dm) = (e_1, \ldots, e_{pf})$,
Let q be a query such that $FV(q) = (q_1, \ldots, q_{pf})$,
$dm \in DM^+(q) \Leftrightarrow FV(dm) \geq FV(q) \quad \Leftrightarrow \forall i: (e_i \geq q_i)$

Lemma 1 is merely a restatement in terms of feature vectors of properties expressed in the definition of $DM^+(q)$. Its proof is left to the reader. Lemma 1 allows us to consider labels for positive or negative examples as a property of feature vectors, by stating that the label of a data mapping depends entirely on its feature vector. This observation is stated formally in Lemma 2.

Lemma 2:
Let dm and dm' be two data mappings such that FV(dm) = FV(dm').
For any query q, $dm \in DM^+(q) \Leftrightarrow dm' \in DM^+(q)$

Lemma 2 is a direct corollary of Lemma 1.

5.3 Feature Vector Label

Each feature vector can always be assigned exactly one of three labels: *pos* (positive), *neg* (negative), or *mis* (missing).

Definition: *Correctly labeled pair*

A *labeled pair* is a pair (fv, lbl), where fv is a feature vector and lbl is one of the three labels {*neg*, *pos*, *mis*}. Let q be a query, (fv, lbl) is a *correctly labeled pair* with respect to q if and only if the following conditions are met:

- If for all data mapping dm in DM, FV(dm) ≠ fv, then lbl = *mis*
- If there exists a data mapping dm such that FV(dm) = fv and dm∈ DM^+(q) then lbl = *pos*
- If there exists a data mapping dm such that FV(dm) = fv and dm∉ DM^+ (q) then lbl = *neg*

Lemma 1 and Lemma 2 guarantee that the three cases in the above definition are mutually exclusive, and that their disjunction is always true.

5.4 Learning Algorithm Sequences

Interaction between the user and the system, described in Section 5, results in the user constructing a label sequence, one labeled pair at a time.

Definition: *Label Sequence*

A label sequence is a sequence of labeled pairs.

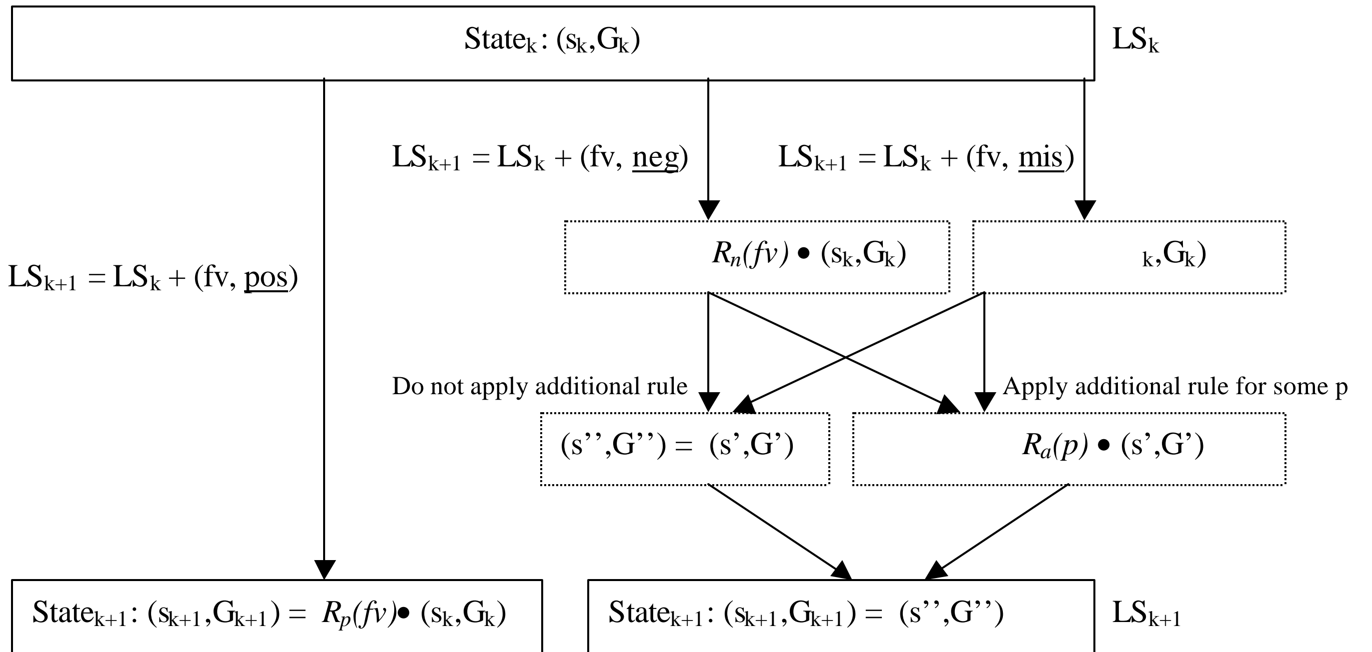


Figure 12 – SPHINX learning algorithm State Transitions

Two quantities characterize the state of the SPHINX learning algorithm itself: the Version Spaces state, and a Label Sequence. Figure 8 illustrates the state transition, when a labeled pair (fv, label) is added to the label sequence. The initial state is characterized by vector (s_k, G_k), and label sequence LS_k. The algorithm branches on the label value, and applies some

rules, which reduce the space of remaining hypothesis by operating on the Version Spaces state. As shown in Figure 6, three different rules exist and therefore three different kinds of operators exist which operate on the Version Spaces state: the *positive rule* with operator $R_p(fv)$ which is applied in the positive branch, the *negative rule* with $R_n(fv)$ which is applied in the negative branch, and the *additional rule* with $R_a(p)$ which can be applied in both the negative and the missing branch.

Definition: *Rule Sequence*

A rule sequence is a sequence of rule operators, which are of three kinds: $R_p(fv)$, $R_n(fv)$, $R_a(p)$.

Definition: Rule sequence *triggered* by a label sequence.

A rule sequence RS is triggered by a label sequence LS, if the rule sequence is the set of actions taken by the Version Spaces algorithm in response to the label sequence LS.

Definition: *Version Set*

Let RS be a rule sequence. The version set VS(RS) is defined inductively as:

- $VS(\emptyset) = QS(s_0, G_0)$: the empty rule sequence gives a version set equal to the whole search space.
- $VS(RS) = QS(s_k, G_k) \Rightarrow VS(RS + R_x(y)) = QS(R_x(y)(s_k, G_k))$: adding a new rule to the rule sequence is equivalent to applying the rule operator to reduce the space of remaining hypothesis.

5.5 Correctness for Positive rule operator

Lemma 3 states that the *positive rule* operator $R_p(fv)$ removes q from the space of remaining hypothesis (QS) if and only if no data mapping with feature vector fv can be a positive example for q. Thus operator $R_p(fv)$ only removes from the space of remaining hypothesis those queries which are incompatible with the correctly labeled pair (fv,pos).

Lemma 3:

Let (fv,pos) be a labeled pair, q a query in QS(s, G) and QS' = $R_p(fv)$(QS):

$$q \notin QS' \Leftrightarrow \forall dm : FV(dm) = fv \Rightarrow dm \notin DM^+(q)$$

A proof for Lemma 3 can be found in the Appendix.

5.6 Correctness for Negative rule operator

Lemma 4 states that the operator $R_n(FV(dm))$ removes q from the space of remaining hypothesis (QS) if and only if dm is not a negative example for DPM(q).

Lemma 4:

Let (fv,pos) be a labeled pair, q a query in QS(s, G) and QS' = $R_n(fv)$(QS):

$$q \notin QS' \Leftrightarrow (\forall dm : FV(dm) = fv \Rightarrow dm \in DM^+(q))$$

A proof for Lemma 4 can be found in the Appendix.

5.7 Correctness for Additional rule operator

Lemma 5 makes explicit the conditions under which two target queries may define the same target view. In doing so Lemma 5 proves that the *Additional rule operators* only remove from the Version Spaces state redundant target query definition. Lemma 5 guarantees that each view, which could still be the correct *target view*, preserves at least one representative query in the Query Set.

Definition: *Compatible with*

A query q is *compatible with* a label sequence LS if and only if the following properties are true:

- $(fv, pos) \in LS \Rightarrow (fv, pos)$ is a correctly labeled pair with respect to q
- $(fv, neg) \in LS \Rightarrow (fv, neg)$ is a correctly labeled pair with respect to q

Lemma 5(k):

If:

$LS_k = ((dm_1, l_1), \ldots, (dm_k, l_k))$ is a label sequence,

and RS_k is a rule sequence triggered by LS_k such that $(R_a(p_1), R_a(p_2), \ldots, R_a(p_a))$ is the exact subsequence of applications of the *additional rule* in RS_k,

and $VS(RS_k) = QS(s_k, G_k)$,

and q is compatible with LS_k,

and q' a query such that $(\forall j \leq a: q_{pj}' = 1) \wedge (\forall i: (\forall j \leq a: i \neq p_j) \Rightarrow (q_i' = q_i))$

Then:

$q' \in VS(RS_k)$

A proof of Lemma 5 can be found in the Appendix.

5.8 Correctness and Termination of the SPHINX learning algorithm.

Theorem 1 guarantees that a query defining the correct target view can always be found in the space of remaining hypothesis maintained by the SPHINX learning algorithm. In particular, if SPHINX has converged to a single query, then that query correctly defines the target view. Theorem 1 is a direct consequence of Lemma 5, and we will leave that proof to the reader.

Theorem 1:

Let $v_t = View(q_t)$ be the target view defined by the target query q_t, LS a label sequence such that q_t is compatible with LS, and RS the rule sequence triggered by LS: $\exists q \in VS(RS)$ such that $View(q) = v_t$.

Theorem 2 states that SPHINX will eventually converge to a single query. To discuss the issue of convergence, and prove Theorem 2, it is necessary to introduce the notion of *partial convergence on a potential feature.*

Definition: *Partial Convergence on a Potential Feature*

Assume the Version Spaces state is (**s**, **G**), with $s = (s_1, s_2, \ldots, s_{pf})$, $G = \{(g_{1,1}, g_{1,2}, \ldots, g_{1,pf}) \ldots (g_{k,1}, g_{k,2}, \ldots, g_{k,pf})\}$. The SPHINX learning algorithm has partially converged for potential feature F_i, if and only if: $s_i = g_{1,i} = g_{2,i} = \ldots = g_{k,i}$.

The reader can verify that if SPHINX has partially converged for all potential features, then it has converged (in the usual sense) and the target query is known.

Theorem 2 states that if each of the 2^{pf} distinct feature vectors is assigned a label, the algorithm is guaranteed to have converged to a solution. The proof figures in the Appendix.

Theorem 2:

Let LS be a label sequence such that LS contains 2^{pf} distinct, correctly labeled pairs, one for every possible feature vector instance. Let RS be the rule sequence triggered by LS, then VS(RS) = QS(s, G) has converged to a single query.

Theorem 1 and Theorem 2 together guarantee eventual convergence of the SPHINX learning algorithm towards a single, correct view definition.

6. Active Learning and Sample Selection

In machine learning, *Active Learning* refers to a process where the system selects examples for the user to label, in order to reduce the cost of labeling unnecessary examples. In this section we look at the problem of selecting the examples that must be submitted to the user.

The number of examples necessary to converge to an answer is an important measure of success for SPHINX, and the goal is to bring the learning algorithm to converge with a minimum number of examples. However, it should be noted that in an adversarial worst-case scenario, the system would be forced to test 2^{pf}-(pf choose 2) negative or missing examples, before the *additional learning rule* could be used to converge. In that far-fetched scenario, the size of the target view, equal to 1, precludes identifying positive examples beyond the initial graphic example. Only negative and missing labels can be added to the label sequence, and only the additional rule operator will make progress towards convergence. In practice, when federating data sources we hope that such *target queries* will be rare, and that the learning algorithm can converge at a more acceptable rate.

6.1 Active Learning as a Search Problem

To explore its search space, SPHINX proceeds incrementally by trying to accomplish a series of intermediate sub-goals. The active learning portion of SPHINX chooses the examples it presents to the user with the purpose of achieving its current sub-goal. Consider the feature vector illustrated in Figure 13. All its feature bits are set to 1 except for F_1. A data mapping instance with this example feature vector would negate only predicate P_1.

F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Figure 13 – Example Feature Vector

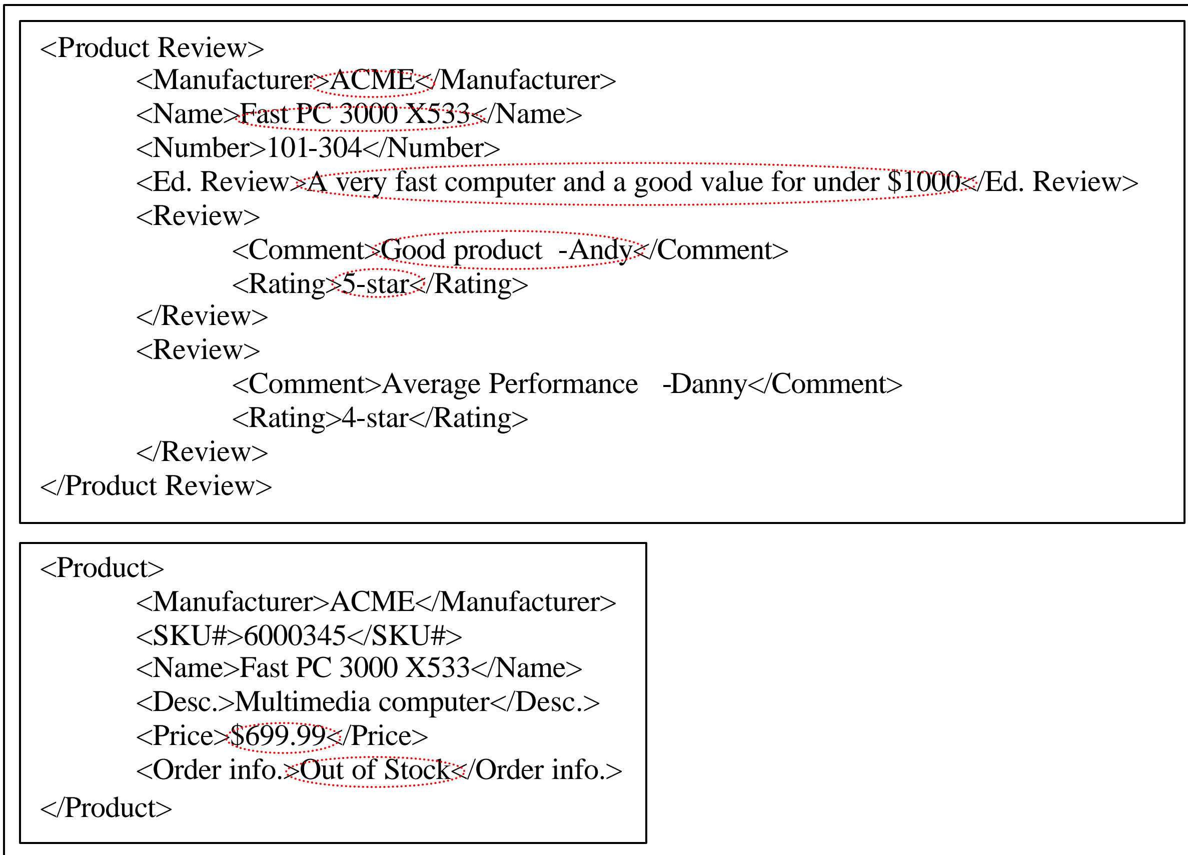


Figure 14 – Data Mapping Instance with the Example Feature Vector shown in Figure 13

Consider the data mapping shown in Figure 14. Its example feature vector is equal to the one shown in Figure 13. This data mapping is similar in all points to the data mapping shown in Figure 2, except for a minor difference in one of the reviews. This minor difference results in the negation of predicate P_1.

If the user assigns this data mapping a positive label then SPHINX will know P_1 cannot be a filter predicate in the target query. Conversely if the user assigns it a negative label then the target query must contain filter predicate P_1. Thus, regardless of the actual label value, SPHINX can ascertain whether or not P_1 belongs to the target query. Provided the same mechanism can be repeated for the other 16 potential features, SPHINX could, in 17 steps, learn the target query.

6.2 Sub-goals for Partial Convergence

We examine one strategy, which proceeds by setting sub-goals for partial convergence, and incrementally leads to full convergence. This strategy seeks to find data mapping instances for each of the 17 sub-goals shown in Figure 15, and then to submit them to the user. To reach these sub-goals, a data mapping instance with example feature vector equal to the sub-goal must be submitted to the user (a query on the source can easily be written to retrieve data mapping instances corresponding to a given feature vector representation). This strategy is particularly attractive, because if the necessary 17 data mappings can be found, regardless of how the user labels them, the algorithm fully converges. However in a practical scenario, using real data sources, none of the data mappings necessary for any of the sub-goals in Figure 15, are likely to

exist. These sub-goals are too specific: they require data mappings with very restrictive integrity constraints.

Sub-goal 1: (0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
determine if P_1 must be included in the target query
Sub-goal 2: (1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
determine if P_2 must be included in the target query
…
…
Sub-goal 17: (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0)
determine if P_{17} must be included in the target query

Figure 15 – Sequence of sub-goals leading to convergence

More generally, because of functional or accidental dependencies in the data and because many predicates will not be independent (e.g. P_1 and P_4) most sub-goals are unreachable and looking for the corresponding data mappings will result in the production of missing labels. Since missing labels do not directly lead to convergence, sub-goals must be chosen carefully.

6.3 Value of Positive Examples vs. Negative Examples

Since a naï ve strategy of proving or disproving each individual feature separately (as in Figure 17) cannot be seriously considered, it is important to understand that the value of a positive example is much higher than the value of a negative example. Consider a sub-goal, which seeks (to prove or disprove) a group of k filter predicates. Finding a positive data mapping example for that sub-goal, will bring the algorithm closer to convergence: by generalizing the most specific feature vector on all k features, the algorithm automatically reaches partial convergence on those k features. On the other hand a negative data mapping example for the same sub-goal does not bring the algorithm much closer to converging, because when all k features are negated at the same time, it is not the case that partial convergence is achieved on any feature; unless the additional learning rule can be applied. Unfortunately the additional learning rule often requires a large amount of labeled pairs in order to be applied.

7. Sample Selection Heuristics

In this Section we look at heuristics to implement sample selection for the SPHINX system. When looking at establishing facts about the target query using sub-goals, we need to consider two factors:

- missing labels are of little value,
- negative labels are also of little value.

Any strategy wishing to minimize user interaction, must focus on finding at least one, possibly several, examples labeled *positive* by the user. Therefore the basis of our strategy is to separate active learning in two phases. In the first phase SPHINX will exclusively focus on proposing data mappings, which it judges are likely to be positive examples. The goal in the first phase is to converge quickly on as many features as possible. In the second phase SPHINX proposes those data mappings that have no particular reason to be positive or negative. Thus the expected convergence rate in the second phase will be minimal, but with the extent of the search space reduced by the first phase, the number of examples can be contained.

7.1 Join Feature Bias

A simple analysis shows that the overwhelming majority of the potential features are selection predicates. A selection predicate is created for every attribute in the schema. A join feature is created only when supported by the initial data-mapping example: a relationship exists between two objects in the data mapping.

		Potential Features	Selection Features	Join Features
Healthcare	Query 1	14	14	0
	Query 2	15	15	0
Sports Statistics	Query 3	25	24	1
	Query 4	25	24	1
5 Star Catalog	Query 5	30	28	2
	Query 6	30	28	2

Table 1 – Selection vs. Join Features

An accidental match between a "9.99" as the price $9.99 and a "9.99" as September 99 is possible, but is an unlikely event given any pair of objects chosen at random by the user. Thus almost all join features observed in the data-mapping example are not flukes and do represent existing semantic relationships. Thus two factors combine here to privilege join features: a pure Cartesian product without a join predicate is a very unlikely operation, and most observed join features between objects belonging to different sets are not accidental.

We elaborate a baseline strategy S_b based on this observation. S_b privileges the search for positive examples in its initial phase by initially never choosing sub-goals negating join predicates. When searching for a sub-goal, strategy S_b looks to negate only a small set of selection predicates (never more than 10). That search is repeated by modifying the set until a positive example is found or the algorithm converges. The small set of negated selection features is chosen with an initial randomization and modified in an incremental search pattern. There is a possibility for backtracking to re-randomize the set, but in the course of running the experiments shown in Table 2, backtracking with S_b occurred only once. Once positive examples have been found, S_b enters its second phase, in which both join and selection predicates will be negated.

It should be observed that in addition to its bias for join features vs. selection features, strategy S_b possesses another built-in bias: it consistently bets that of all the potential features (a large number), only a very small number is likely to actually appear in the *Where* clause of the target query.

7.2 Information Gain Bias

We make the observation that not all features are equally likely. Consider the following feature predicate: "Name = 'Supervac 4000'". It is unlikely to appear in a view defining query since few objects in the source, perhaps only one, will fulfill that predicate, making it useless for any view definition. On the other hand a feature such as "manufacturer/state = 'TX'" is more likely. A significant proportion of the objects may well fall in the 'TX' category and building a view with those might be of use.

We can measure for each selection predicate their information gain. The basic assumption is that the information gain will serve to estimate the likelihood of a predicate.

$$IG(P) = - (|S_1|/(|S_1|+|S_2|))ln(|S_1|/(|S_1|+|S_2|)) - (|S_2|/(|S_1|+|S_2|))ln(|S_2|/(|S_1|+|S_2|))$$

The information gain IG(P) is a function of the filter factor FF(P), and is maximal when FF(P) = 0.5. IG(P) will yield high scores for predicates on enumerated types, and low scores for predicates on infinite types.

We note the information gain metric IG(P), relies on the same catalog statistics used in those query cost models. It has been shown these statistics can be derived even in distributed systems where catalog information is not directly accessible [11,30]. Thus, we are confident we can always rely on such statistics to drive our heuristics.

We introduce a new strategy S_c, a refinement of S_b based on this information gain bias. This strategy does not require likelihood estimations to be accurate. The likelihood function should, above all, cluster predicates into two major categories: the most unlikely predicates (extremely low information gain and infinite domain), and the other predicates (low to high information gain and enumerated domain). This clustering will replace the random process used in S_b.

A comparable bias could be introduced to estimate the likelihood of individual join predicates, however as discussed earlier the small number of join predicates precludes the need.

7.3 Experiments

We implemented a prototype system complete with graphic-user interface. This prototype handles data mapping instances presented here, as well as mappings from meta-data elements to data (in XML, tag-names can be extracted with the getTagName function). This small higher-order generalization allows from a broader range of restructuring queries across schematically disparate sources, without any substantial changes to the overall system.

We chose three domains to experiment with data integration. All of these problems were actual internet database integration tasks conducted under contract, in an ad-hoc fashion by a web services consulting firm. Almost all source data was available only in HTML form, with the source sites wrapped [3,7] to produce structured results. These experiments with SPHINX recreate those schema integration tasks, and are ranked in Table 2, by increasing level of empirical complexity. In the first domain the target queries populate a Healthcare provider directory database. The second domain is based on sport statistics databases. The third domain is the '5 Star Catalog' for electronics and comes from the area of online pricing catalogs for B2B merchandise distributors. A slightly simplified version is used as an illustrative example in this exposition.

Table 2 gathers some results: each test set includes source databases (which can be found at [37]), a target schema, and two interesting target queries populating different tables of the target schema. The number of features (i.e. Selection and Join predicates) appearing in the *Where* clause of each query is shown (e.g. 1J, 0S: 1 Join and no Selection predicates). The number of examples SPHINX requires to reach the target query is averaged over ten runs for each query and shown in Table 2. The decimal averages are a product of random factors present in both heuristics.

Strategy S_b is more successful than S_c for the target queries which do not have a selection predicate in their *Where* clause. The performance of strategy S_b degrades quickly when the target query contains even a single selection predicate. Strategy S_c, shows a more stable behavior, its performance only slowly decreasing when the complexity of the target query increases. This can be attributed to the inability of strategy S_b to differentiate among the selection features, and hence pick the ones, which can be excluded from the target query with high probability.

		Query size	Potential Features	Strategy S_c Number of Examples			Strategy S_b Number of Examples		
				Total	Pos.	Neg.	Total	Pos.	Neg.
Healthcare	Query 1	0J, 0S	14	2.2	2.2	0.0	1.4	1.4	0.0
	Query 2	0J, 1S	15	4.7	2.0	2.7	5.5	2.5	3.0
Sports Statistics	Query 3	1J, 0S	25	4.9	3.7	1.2	2.5	1.5	1.0
	Query 4	1J, 1S	25	4.8	1.8	3.0	11.4	1.6	9.8
5 Star Catalog	Query 5	2J, 0S	30	5.9	3.2	2.7	4.0	2.0	2.0
	Query 6	2J, 1S	30	8.9	2.7	6.2	11.7	2.0	9.7

Table 2 – Experiments with SPHINX

		S_c	S_b	oracle			random		
		Total	Total	Total	Pos.	Neg.	Total	Pos.	Neg.
Healthcare	Query 1	2.2	1.4	1	1	0	26	13	13
	Query 2	4.7	5.5	2	1	1	26	13	13
Sports Statistics	Query 3	4.9	2.5	2	1	1	7.5	3.5	3.0
	Query 4	4.8	11.4	3	1	2	22	11	11
5 Star Catalog	Query 5	5.9	4.0	4	1	3	12	5.5	6.0
	Query 6	8.9	11.7	5	1	4	42	21	21

Table 3 – Active learning vs. passive learning

Table 3 compares the performance of both active learning strategies for SPHINX with two experiments in which SPHINX is hobbled to become a passive learning system. These two experiments do not represent valid strategies but are designed to identify bounds, both lower (oracle) and upper (random) on the number of examples an active learning algorithm may require

In these passive learning experiments, the user carries the burden of constructing data mapping examples as well as labeling them. SPHINX merely indicates when the system has converged to a target query. To measure a lower-bound, an omniscient user, (us), constructed the optimal sequence of examples to converge SPHINX as quickly as possible. To measure an upper-bound a naive user chooses examples at random from the set DM. At each step there is an equal probability of a positive or a negative example being chosen. Each example is picked randomly from its respective population of positive (DM^+) or negative examples ($DM-DM^+$) with equal probability. Unlike the oracle this user is not in a feedback loop with the learning system, and does not know which examples need to be picked next in order to finish converging the system.

These experiments show that even with imperfect heuristics, the observed complexity is correlated with the size and complexity of the target query rather than with the number of potential features. We can observe both that the number of required examples spikes when predicates are added to the target query, and that the most simple target queries require a small number of examples, even when the number of potential features is large.

8. Conclusion

We built a Version Spaces model for query discovery by example and developed the SPHINX learning algorithm, by adding a new kind of label and a new learning rule to the two labels and the two rules in the original Version Spaces algorithm. This new algorithm allows full and accurate verification by a user of potential mappings of semantic relationships. This is accomplished entirely by example, where only the initial data-mapping example needs to be supplied by the user. The active learning and sample selection system incorporated in SPHINX generate additional examples, which are labeled positive or negative by the user. We present an effective search heuristic which minimizes the number of such examples submitted to the user by quickly eliminating potential features.

We note that SPHINX, has no real intelligent understanding of the semantic properties of data. Rather it focuses on a purely syntactic understanding of the federating views and on a user interaction model. SPHINX relies entirely upon the user to provide semantic knowledge and does not seek to derive it from context and from existing information. This split between user and system responsibilities was motivated by our specific goals. However, future work could incorporate existing machine learning techniques, which exploit such information. Without abandoning the goal of accuracy, the system can make more educated guesses to improve current heuristics and converge quicker in complex situations.

With SPHINX we grapple with the issue of designing a GUI Interface for complex data transformations. There already are successful GUIs for XML tag matching, and with the typical size of a DTD, these represent the only realistic way for most people to interact with XML mappings. To integrate data from multiple XML types, and to perform more complex operations such as un-nesting and joins, it would be desirable to incorporate automated schema matching tools into those simple GUI paradigms. However, as those semantic tools currently require an expert to check correctness, with SPHINX we sought to identify a first set of more advanced semantic elements which could still be brought to the level of a GUI for a non-technical user.

Future work should also address a wider range of practical issues in schema integration within the framework we established. In particular, we leave open the issue of adjusting and testing the current SPHINX strategy to handle different kinds of queries, such as aggregation operators and disjunction, of incorporating Skolem functions for objects, and of explicitly handling more complex XML such as DAGs. Broadening the generality of SPHINX in this fashion will mean expanding the current GUI principles to require several kinds of user input beyond our initial minimalist approach.

Bibliography

[1] Abiteboul S., S. Cluet, T. Milo: Correspondence and Translation for Heterogeneous Data. ICDT 1997: 351-363

[2] Barbançon F., D. Miranker: Implementing Federated Databases Systems by Compiling SchemaSQL. IDEAS 2002: 192-201.

[3] Baumgartner R., S. Flesca, G. Gottlob: Visual Web Information Extraction with Lixto. VLDB 2001: 119-128.

[4] Bergamaschi S., S. Castano, M. Vincini: Semantic Integration of Semistructured and Structured Data Sources. SIGMOD Record 28(1), 1999, 54-59.

[5] Castano S., De Antonelli V.: A schema analysis and reconciliation tool environment. IDEAS 1999: 53-62.

[6] Cluet S., C. Delobel, J. Siméon, K. Smaga: Your Mediators Need Data Conversion ! SIGMOD 1998: 177-188

[7] Crescenzi V., G. Mecca, P. Merialdo: RoadRunner: Towards Automatic Data Extraction from Large Web Sites. VLDB 2001: 109-118
[8] Doan A., P. Domingos, A. Halevy: Reconciling Schemas of Disparate Data Sources: A Machine Learning Approach. SIGMOD Conference 2001.
[9] Fernandez M., A. Morishima, D. Suciu: Efficient Evaluation of XML Middle-ware Queries. SIGMOD 2001
[10] Garcia-Molina H., Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, J. Widom: The TSIMMIS Approach to Mediation: Data Models and Languages. JIIS 8(2): 117-132 (1997)
[11] Haas L., D. Kossman, E. Wimmers, J. Yang: Optimizing Queries Across Diverse Data Sources. VLDB 1997: 276-285
[12] Hirsh H.: Theoretical Underpinnings of Version Spaces. IJCAI 1991, 665-670.
[13] Kent W.: Solving Domain Mismatch and Schema Mismatch Problems with an Object-Oriented Database Programming Language. VLDB 1991: 147-160
[14] Krishnamurthy R., W. Litwin, W. Kent: Language Features for Interoperability of Databases with Schematic Discrepancies. SIGMOD Conference 1991: 40-49.
[15] Lakshmanan L., F. Sadri, I. Subramanian: SchemaSQL - A Language for Interoperability in Relational Multi-Database Systems. VLDB 1996: 239-250
[16] Levy A., A. Rajaraman, J. Ordille: Querying Heterogeneous Information Sources Using Source Descriptions. VLDB 1996: 251-262.
[17] Li W., C. Clifton, S. Liu: SemInt: a tool for identifying attribute correspondences in heterogeneous databases using neural network. Data and Knowledge Engineering 33(1): 49-84 (2000).
[18] Madhavan J., P. Bernstein, E. Rahm: Generic Schema Matching with Cupid. VLDB 2001: 49-58
[19] Manolescu I., D. Florescu, D. Kossmann: Answering XML Queries on Heterogeneous Data Sources. VLDB 2001: 241-250
[20] Miller R., L. Haas, M. Hernández: Schema Mapping as Query Discovery. VLDB 2000: 77-88
[21] Milo T., S. Zohar: Using Schema Matching to Simplify Heterogeneous Data Translation. VLDB 1998: 122-133.
[22] Mitchell T.: Version Spaces: A Candidate Elimination Approach to Rule Learning. IJCAI 1977: 305-310
[23] Mitra P., G. Wiederhold, M. Kersten: A Graph-Oriented Model for Articulation of Ontology Interdependencies. EDBT 2000: 86-100
[24] Palopoli L., G. Terracina, D. Ursino: The System DIKE: Towards the Semi-Automatic Synthesis of Cooperative Information Systems and Data Warehouses. ADBIS-DASFAA 2000: 108-117.
[25] Park Y., Han Y., Choi K.: Automatic Thesaurus Construction Using Bayesian Networks. CIKM 1995: 212-217.
[26] Rahm E., P. Bernstein: A survey of approaches to automatic schema matching. VLDB Journal 10(4): 334-350 (2001).
[27] Roth M., M. Arya, L. Haas, M. Carey, W. Cody, R. Fagin, P. Schwarz, J. Thomas II, E. Wimmers: "The Garlic Project". SIGMOD 1996: 557.
[28] Shanmugasundaram J., J. Kiernan, E. Shekita, C. Fan, J. Funderburk: Querying XML Views of Relational Data. VLDB 2001: 261-270
[29] Takenobu T., Makoto I., Hozumi T.: Automatic Thesaurus Construction Based on Grammatical Relations. IJCAI 1995:1308-1313.
[30] Tomasic A., L. Raschid, P. Valduriez: Scaling Heterogeneous Databases and the Design of Disco. ICDCS 1996: 449-457

[31] Vassalos V., Y. Papakonstantinou: Describing and Using Query Capabilities of Heterogeneous Sources. VLDB 1997: 256-265.
[32] Vidal M., L. Raschid, J. Gruser: A Meta-Wrapper for Scaling up to Multiple Autonomous Distributed Information Sources. CoopIS 1998: 148-157
[33] XML Query: http://www.w3.org/TR/XQuery
[34] Yan L., M. Özsu, L. Liu: Accessing Heterogeneous Data Through Homogenization and Integration Mediators. CoopIS 1997: 130-139
[35] Yan L., R. Miller, L. Haas, R. Fagin: Data Driven Understanding and Refinement of Schema Mappings. SIGMOD Conference 2001.
[36] Zloof M.: Query-by-Example: A Data Base Language. IBM Systems Journal, 1977.
[37] http://www.cs.utexas.edu/users/francois/databases.html

Appendix

Formal proofs are included in this Appendix for the referees' consideration.

Lemma 3:

Let (fv,pos) be a labeled pair, q a query in QS(s, G) and QS' = $R_p(fv)$(QS):

$$q \notin QS' \Leftrightarrow (\forall dm : FV(dm) = fv \Rightarrow dm \notin DM^+(q))$$

<u>Proof:</u>

Assume fv = $(e_1, e_2, \ldots, e_{pf})$, q = $(q_1, q_2, \ldots, q_{pf})$
QS = (s, G), QS' = (s', G')
G = $\{(g_{1,1}, \ldots, g_{1,pf}),$
$(g_{2,1}, \ldots, g_{2,pf}),$
$\ldots$
$(g_{k,1}, \ldots, g_{k,pf})\}$

- $q \notin QS' \Rightarrow (\forall dm : FV(dm) = fv \Rightarrow dm \notin DM^+ (q))$

Assume dm such that FV(dm) = $(e_1, e_2, \ldots, e_{pf})$.
$q \in QS, q \notin QS' \quad \Leftrightarrow (\forall i: q_i \le s_i) \wedge (\exists i: g_i \le q) \wedge ((\exists i: q_i > s_i') \vee (\forall i: \neg(g_i' \le q)))$
and since $\forall i: g_i = g_i'$:
$\Leftrightarrow (\forall i: q_i \le s_i) \wedge (\exists i: q_i > s_i') \wedge (\exists i: g_i \le q)$
$\Leftrightarrow (\exists i: s_i \ge q_i > s_i') \wedge (q \in QS)$
and since $s_i > s_i'$, we must have $s_i = 1$ and $s_i' = 0$, which by definition of s' and R_p implies $e_i=0$:
$\Rightarrow (\exists i: q_i > e_i)$
$\Rightarrow dm \notin DM^+ (q)$

- $(\forall dm : FV(dm) = fv \Rightarrow dm \notin DM^+ (q)) \Rightarrow q \notin QS'$

Assume dm such that FV(dm) = $(e_1, e_2, \ldots, e_{pf})$.
$dm \notin DM^+ (q) \quad \Rightarrow (\exists i: q_i > e_i)$
$\Leftrightarrow (\exists i: q_i = 1 \wedge e_i = 0)$
and since $q \in QS$, we have $(\forall i: q_i \le s_i) \wedge (\exists i: g_i \le q)$
$\Rightarrow (\exists i: q_i = 1 \wedge e_i = 0 \wedge q_i \le s_i)$
$\Rightarrow (\exists i: q_i = 1 \wedge e_i = 0 \wedge s_i = 0)$
and since by definition of s' and R_p, s_i' must be 0 in that case :
$\Rightarrow (\exists i: q_i = 1 \wedge s_i = 1 \wedge s_i' = 0)$
$\Rightarrow (\exists i: q_i > s_i')$
$\Rightarrow q \notin QS'$

■

Lemma 4:

Let (fv,pos) be a labeled pair, q a query in QS(s, G) and QS' = $R_n(fv)$(QS):

$$q \notin QS' \Leftrightarrow (\forall dm : FV(dm) = fv \Rightarrow dm \in DM^+ (q))$$

<u>Proof:</u>

Assume $fv = (e_1, e_2, \ldots, e_{pf})$, $q = (q_1, q_2, \ldots, q_{pf})$
$QS = (s, G)$, $QS' = (s'', G'')$
$s'' = s$
$G = \{(g_{1,1}, \ldots, g_{1,pf}),$
$(g_{2,1}, \ldots, g_{2,pf}),$
$\ldots$
$(g_{k,1}, \ldots, g_{k,pf})\}$
$G'' = \{ (g_{i,1}', \ldots, g_{i,pf}') \mid [\ (g_{i,1}', \ldots, g_{i,pf}') \leq s'\ ]$ **Ù**
$[\ \ [(\exists p: (g_{p,1}, \ldots, g_{p,pf}) \in G) \wedge (\forall j: e_j=0 \Rightarrow g_{p,j} = 0)) \wedge (\exists f: (\forall j \neq f: g_{i,j}' = g_{p,j}) \wedge e_f=0 \wedge g_{i,f}' = 1)]$
$\vee [(\exists p: (g_{p,1}, \ldots, g_{p,pf}) \in G) \wedge (\exists f: e_f=0 \wedge g_{p,f} = 1) \wedge (\forall j: g_{p,j} = g_{i,j}')]\ \ ]\ \}$

- $q \notin QS' \Rightarrow (\ \forall dm : FV(dm) = fv \Rightarrow dm \in DM^+(q)\ )$

Assume dm such that $FV(dm) = (e_1, e_2, \ldots, e_{pf})$
since $s'' = s$: $q \notin QS' \Rightarrow (\exists j\ \forall i: q_i \geq g_{j,i}) \wedge (\forall j\ \exists i: q_i < g_{j,i}')$.
Let z be such that $\forall i: q_i \geq g_{z,i}$
Assume A: $(\exists i: e_i = 0 \wedge g_{z,i} = 1)$
in that case $g_z \in G''$ (by fulfilling the second part of the disjunction),
and because $g_z \leq q \leq s'' = s$, we find that $q \in QS'$ which is impossible.
We therefore deduce $\neg$A: $(\forall i: (e_i = 0) \Rightarrow (g_{z,i} = 0))$
Let f be any f such that $e_f = 0$, we define the vector $ng = (ng_1, ng_2, \ldots, ng_{pf})$
with $ng_f = 1$, and $(\forall i \neq f: ng_i = g_{z,i})$
Assume B: $s_f = 0$,
then because $q \leq s$, we have $q_f = 0$
Assume $\neg$B: $s_f = 1$,
then by definition $ng \in G'$
and since $(\forall i \neq f: ng_i = g_{z,i})$, we have: $(\forall i \neq f: ng_i \leq q_i)$
Assume C: $q_f = 1$,
then $ng_f \leq q_f$,
and since $(\forall i: ng_i \leq q_i)$, therefore $ng \leq q \leq s'' = s$
this implies $q \in QS'$ which is impossible.
We can therefore deduce $\neg$C: $q_f = 0$.
QED (we have just proved $\forall f: e_f = 0 \Rightarrow q_f = 0$)

- $(\ \forall dm : FV(dm) = fv \Rightarrow dm \in DM^+(q)\ ) \Rightarrow q \notin QS'$

Assume dm such that $FV(dm) = (e_1, e_2, \ldots, e_{pf})$.
$\in G''$: $g'' = (g_{z,1}', g_{z,2}', \ldots, g_{z,pf}')$ with all the properties listed above for a member of G'',
in particular $(\exists p: g_p=(g_{p,1}, \ldots, g_{p,pf}) \in G)$ such that:
$[(\forall j: e_j=0 \Rightarrow g_{p,j} = 0)) \wedge (\exists f: (\forall j \neq f: g_{i,j}' = g_{p,j}) \wedge e_f=0 \wedge g_{i,f}' = 1)]$
$\vee\ \ [(\exists f: e_f=0 \wedge g_{p,f} = 1) \wedge (\forall j: g_{p,j} = g_{i,j}')]$
Take such a p:
Assume A: $(q \geq g_p)$
and since $dm \in PDM(q)$, therefore we have $(\forall i: e_i \geq q_i \geq g_{p,i})$.
The property $(\exists f: e_f = 0 \wedge g_{p,f} = 1)$ is now impossible,
therefore by definition of G'': $(\exists f: (e_f = 0) \wedge (g_{z,f}' = 1) \wedge (\forall i \neq f: g_{z,i}' = g_{p,i}))$.
Take such an f:
because $(e_f = 0)$ and $(\forall i: e_i \geq q_i)$
the only possibility is in that case: $q_f = e_f = 0$

$(q_f = 0)$ and $(g_{z,f}' = 1)$ imply $\neg(g'' \leq q)$

Assume $\neg$A: $(\exists i: q_i < g_{p,i})$

Assume B: $(\forall j: g_{z,j}' = g_{p,i})$

then $(\exists i: q_i < g_{p,i} = g_{z,j}')$ which implies $\neg(g'' \leq q)$

Assume $\neg$B: $\neg(\forall j: g_{z,j}' = g_{p,i})$

Then by definition of G'', the other part of the disjunction must be true:

$[(\forall j: e_j{=}0 \Rightarrow g_{p,j} = 0)) \wedge (\exists f: (\forall j \neq f: g_{z,j}' = g_{p,j}) \wedge e_f{=}0 \wedge g_{z,f}' = 1)]$

in particular: $(\exists f: g_{z,f}' = 1 \wedge (\forall j \neq f: g_{z,j}' = g_{p,j}))$

therefore: $(\forall j: g_{z,j}' \geq g_{p,j})$.

Recall that $(\exists i: q_i < g_{p,i}) \Rightarrow (\exists i: q_i < g_{p,i} \leq g_{z,i}') \Rightarrow \neg(g'' \leq q)$

QED (we have just proved $(\forall g'' \in G'': \neg(g'' \leq q))$)

■

Lemma 5(k):

If:

$LS_k = ((fv_1, l_1), \ldots, (fv_k, l_k))$ is a label sequence,

and RS_k is a rule sequence triggered by LS_k such that $(R_a(p_1), R_a(p_2), \ldots, R_a(p_a))$ is the exact subsequence of applications of the *additional rule* in RS_k,

and $VS(RS_k) = QS(s_k, G_k)$,

and q is compatible with LS_k,

and q' a query such that $(\forall j \leq a: q_{pj}' = 1) \wedge (\forall i: (\forall j \leq a: i \neq p_j) \Rightarrow (q_i' = q_i))$

Then:

$q' \in VS(RS_k)$

Proof:

Lemma 5 is parameterized by k, the length of the label sequence. The proof is an induction on k.

- $k = 0$

$LS_0 = \emptyset$, $RS_0 = \emptyset$, $a=0$

We simply verify that q = q' and that both are in $VS(\emptyset) = QS(s_0, G_0)$ which is the whole search space.

- Assume Lemma 5(k) is true: prove Lemma 5(k+1)

Case 1:

The subsequence of applications of *additional rule operators* is the same for RS_{k+1} and RS_k. In other terms there is no application of the *additional rule operator* between step k and step k+1.

Assume q and q' are queries such that q is compatible with LS_{k+1} and q' is such that $(\forall j \leq a: q_{pj}' = 1) \wedge (\forall i: (\forall j \leq a: i \neq p_j) \Rightarrow (q_i' = q_i))$.

We can apply the induction hypothesis, Lemma 5(k) on LS_k, RS_k, q and q': therefore $q' \in VS(RS_k)$.

Assume that $fv = (e_1, e_2, \ldots, e_{pf})$.

There are three further cases on the value of the label l_{k+1}:

- If l_{k+1} = pos, then there exists $dm_{k+1} \in DM^+(q)$, such that $FV(dm_{k+1}) = fv_{k+1}$ because q is compatible with LS_{k+1}.

  $dm_{k+1} \in PDM(q) \Rightarrow (\forall i: e_i \geq q_i)$

  There are two further cases:

  - Let i be such that $(\forall j \leq a: i \neq p_j)$, then $q_i' = q_i$ and $e_i \geq q_i'$
  - Let i be such that $(\exists j \leq a: i = p_j)$, then

    let j be such that $j \leq a$ and $i = p_j$

    Assume A: $(e_i = 0)$, then $fv_{k+1} \in 0_i$,

then Precondition for $R_a(i=p_j)$ in the rule sequence, dictates that the label l_{k+1} be negative or missing. This is impossible.

We can therefore deduce $\neg A$: ($e_i = 1$), and $e_i \geq q_i$' is assured.

Thus we proved with both cases that ($\forall i$: $e_i \geq q_i$'),
and therefore that $dm_{k+1} \in$ PDM(q').
Using Lemma 3, we can deduce that q' $\in R_p(fv_{k+1})(s_k, G_k)$,
therefore q' $\in$ VS(RS_{k+1}).

- If l_{k+1} = neg, then there exists $dm_{k+1} \notin$ PDM(q) such that FV(dm_{k+1}) = fv_{k+1}, because q is compatible with LS_{k+1}
$dm_{k+1} \notin DM^+(q) \Rightarrow (\exists i: e_i < q_i)$.
Let i be such that $e_i < q_i$. There are two further cases:
    - ($\forall j \leq a$: $i \neq p_j$), then $q_i = q_i$' and $e_i < q_i$'. Therefore $dm_{k+1} \notin$ PDM(q')
    - ($\exists j \leq a$: $i = p_j$), then let j be such that $j \leq a$ and $i = p_j$.
    $e_i < q_i \Rightarrow e_i = 0$, and since q_i' = 1, $e_i < q_i$ is assured. Therefore $dm_{k+1} \notin$ PDM(q').

  With both cases we established $dm_{k+1} \notin$ PDM(q').
  Using Lemma 4, we can deduce that q' $\in R_n(fv_{k+1})(s_k, G_k)$,
  and since we are in the case where there is no application of the *additional rule* between step k and step k+1: q' $\in$ VS(RS_{k+1})
- If l_{k+1} = mis, then since there is no application of the *additional rule*, $RS_k = RS_{k+1}$ and q' $\in$ VS(RS_{k+1})

<u>Case 2:</u>
The subsequence of applications of the *additional rule operator* is incremented from RS_k to RS_{k+1} by the application of $R_a(p_{a+1})$. In other terms $R_a(p_{a+1})$ is applied between step k and step k+1.
Assume q and q' are queries such that q is compatible with LS_{k+1} and q' is such that ($\forall j \leq a+1$: q_{pj}' = 1) $\wedge$ ($\forall i$: ($\forall j \leq a+1$: $i \neq p_j$) $\Rightarrow$ (q_i' = q_i)).
Let q'' = (q_1'', q_2'', …, q_{pf}'') be such that ($\forall j \leq a$: q_{pj}'' = 1) $\wedge$ ($\forall i$: (($\forall j \leq a$: $i \neq p_j$) $\Rightarrow$ q_i'' = q_i)).
We can apply the inductive hypothesis, Lemma 5(k) to q'': q'' $\in$ VS(RS_k).
Assume that $fv_{k+1} = (e_1, e_2, …, e_{pf})$.
There are two further cases:

- If l_{k+1} = neg, then there exists $dm_{k+1} \notin$ PDM(q) such that FV(dm_{k+1}) = fv_{k+1}, because q is compatible with LS_{k+1}
$dm_{k+1} \notin DM^+(q) \Rightarrow (\exists i: e_i < q_i)$
There are two further cases:
    - Let i be such that ($\forall j \leq a$: $i \neq p_j$), then q_i'' = q_i and therefore $e_i < q_i$''
    - Let i be such that ($\exists j \leq a$: $i = p_j$), then
    let $j \leq a$ such that $i = p_j$: in that case q_i'' = 1
    and since $e_i < q_i$. The only possibility is $e_i = 0 < q_i$'' = 1.

  Thus we proved with both cases that ($\exists i$: $e_i < q_i$'')
  $\Rightarrow dm_{k+1} \notin$ PDM(q'').
  Using Lemma 4: q'' $\in R_n(fv_{k+1})(s_k, G_k)$.
  Note that ($\forall i \neq p_{a+1}$: q_i'' = q_i') $\wedge$ (q_{pa+1}' = 1).
  There are two cases:
    - <u>Case A:</u> q_{pa+1} = 1 : in this case since q_{pa+1}''=q_{pa+1} , we have q'' = q',
    and since q_{pa+1}'' = 1, by definition of the action for $R_a(p_{a+1})$:
    q'' $\in R_a(p_{a+1})(R_n(fv_{k+1})(s_k, G_k))$
    $\in$ VS(RS_{k+1}).
    - <u>Case B:</u> q_{pa+1} = 0 : q_{pa+1}'' = 0, q_{pa+1}' = 1

We will prove that q’ is compatible with LS_{k+1}, then we will deduce that q’ $\in$ $VS(RS_{k+1})$.

- Let $i \leq k+1$ be such that $(fv_i, l_i = pos) \in LS_k$.
  There exists $dm_i \in PDM(q'')$ s.t. $FV(dm_i) = fv_i = (f_1, \ldots, f_{pf})$
  $dm_i \in VM(q'') \Rightarrow (\forall i: f_i \geq q_i'')$.
  If $f_{pa+1} = 1$ then $(\forall i: f_i \geq q_i') \Rightarrow dm_i \in PDM(q')$
  If $f_{pa+1} = 0$ then by the precondition for $R_a(p_{a+1})$, l_i is neg or mis, which is impossible.
  Therefore $dm_i \in PDM(q')$.
- Let $i \leq k+1$ be such that $(fv_i, l_i = neg) \in LS_k$
  There exists $dm_i \notin DM^+(q'')$ s.t. $FV(dm_i) = fv_i = (f_1, \ldots, f_{pf})$
  In this case $dm_i \notin PDM(q'')$, which implies that $\exists i: f_i < q_i''$.
  If $i = p_{a+1}$ then since $q_{pa+1}'' = 0$, there is a contradiction.
  If $i \neq p_{a+1}$. In that case: $q_i'' = q_i'$, which implies that $f_i < q_i''=q_i'$.
  Therefore $\exists i: f_i < q_i'$ and $dm_i \notin DM^+(q')$.

We have just established that q’ is compatible with LS_{k+1}.
By induction on the label sequence LS_k, we now prove that q’ $\in$ $VS(RS_{k+1})$:

- q’ $\in VS(\emptyset)$. q’ is in the initial search space.
- assume q’ $\in VS(RS_i) = QS(s_i, G_i)$, RS_i triggered by LS_i, $LS_{i+1} = LS_i + (fv_i, l_i)$
  - If l_i = pos, since q’ is compatible with LS’,
    there exists $dm_i \in DM^+(q')$ s.t. $FV(dm_i) = fv_i$
    $\in VS(RS_i + R_p(fv_i))$
  - If l_i = neg, since q’ compatible with LS’,
    there exists $dm_i \notin PDM(q')$ s.t. $FV(dm_i) = fv_i$
    by Lemma 4, q’ $\in R_n(fv_i,)(s_i, G_i)$.
    $\forall j \leq a+1$: $q_{pj}' = 1$ and by definition of the $R_a(p_j)$: ($\forall j$: q’ $\in R_a(p_j)(R_n(fv_i)(s_i, G_i))$.
    Since by definition of the algorithm, when l_i = neg, RS_{i+1} is equal to either $RS_i+R_n(fv_i)$ or $RS_i+R_n(fv_i)+R_a(p_j)$ for some j:
    $\in VS(RS_{i+1})$.
  - If l_i=mis, then
    $\forall j \leq a+1$, $q_{pj}' = 1$, and by definition of the $R_a(p_j)$ operator, we have $\forall j$: q’ $\in R_a(p_j)(s_i, G_i)$.
    Since by definition of algorithm, RS_{i+1} is equal to either RS_i or $RS_i+R_a(p_j)$ for some j:
    q’ $\in VS(RS_{i+1})$.

- If l_{k+1} = mis, there are two cases:
  - If $q_{pa+1} = 1$, then q’ = q’’.
    q’ = q’’ $\in VS(RS_k)$,
    and since q_{pa+1}'=1, by definition of $R_a(p_{a+1})$:
    q’ $\in R_a(p_{a+1})(s_k, G_k) = VS(RS_{k+1})$.
  - If $q_{pa+1}' = 0$. Same scenario and same proof as in Case B above. We first prove that q’ is compatible with LS_{k+1}, then by a mini-induction that q’ $\in VS(RS_{k+1})$. QED

■

Theorem 2:

Let LS be a label sequence such that LS contains 2^{pf} distinct, correctly labeled pairs, one for every possible feature vector instance. Let RS be the rule sequence triggered by LS, then VS(RS) = QS(s, G) has converged to a single query.

Proof:
In order to prove that VS(RS) has converged to a single query, we will prove that partial convergence has been reached for each of the potential features.
Let $f \leq pf$, be a potential feature:

- Assume A: $[\forall i: (((dm_i = (e_1, e_2, \ldots, e_{pf}), l_i) \in LS) \wedge (e_f = 0)) \Rightarrow l_i = neg \text{ or } mis]$
  Since LS contains all possible feature vectors, the precondition for $R_a(f)$ is fulfilled. Therefore $R_a(f)$ is guaranteed to be in the rule sequence RS and by definition of $R_a(f)$: $(\forall i: s_f = g_{i,f} = 1)$.
  The algorithm has partially converged on feature f.
- Assume $\neg$A: $[\exists i: (((dm_i = (e_1, e_2, \ldots, e_{pf}), l_i) \in LS) \wedge (e_f = 0)) \Rightarrow l_i = pos]$
  In that case there exists i such that $R_p(FV(dm_i))$ is in the rule sequence RS.
  Since $e_f = 0$, by definition of $R_p(FV(dm_i))$: $(\forall i: s_f = g_{i,f} = 0)$.
  The algorithm has partially converged on feature f.

Thus the algorithm has converged on all potential features, and $(\forall i: s_f = g_{i,f})$

■