Automatic Integration of Web Search Interfaces with WISE-Integrator

Hai He[1] , **Weiyi Meng**[1], **Clement Yu**[2], **Zonghuan Wu**[3]

[1] *Department of Computer Science, SUNY at Binghamton, Binghamton, NY 13902, USA*
{haihe, meng}@cs.binghamton.edu

[2] *Department of Computer Science, Univ. of Illinois at Chicago, Chicago, IL 60607, USA*
yu@cs.uic.edu

[3] *Center for Adv. Compu. Studies, Univ. of Louisiana at Lafayette, Lafayette, LA 70504, USA*
zwu@cacs.louisiana.edu

Abstract. An increasing number of databases are becoming Web accessible through form-based search interfaces, and many of these sources are database-driven E-commerce sites. It is a daunting task for users to access numerous Web sites individually to get the desired information. Hence, providing a unified access to multiple E-commerce search engines selling similar products is of great importance in allowing users to search and compare products from multiple sites with ease. One key task for providing such a capability is to integrate the Web search interfaces of these E-commerce search engines so that user queries can be submitted against the integrated interface. Currently, integrating such search interfaces is carried out either manually or semi-automatically, which is inefficient and difficult to maintain. In this paper, we present WISE-Integrator – a tool that performs automatic integration of **W**eb **I**nterfaces of **S**earch **E**ngines. WISE-Integrator explores a rich set of special meta-information that exists in Web search interfaces, and employs the information to identify matching attributes from different search interfaces for integration. It also resolves domain differences of matching attributes. In this paper, we also discuss how to automatically extract information from search interfaces that is needed by WISE-Integrator to perform automatic interface integration. Our experimental results based on 143 real-world search interfaces in four different domains indicate that WISE-Integrator can achieve high attribute matching accuracy and can produce high-quality integrated search interfaces without human interactions.

Keywords: Web search interface integration, Schema integration, Attribute matching, Interface Extraction, Metasearch

1 Introduction

With the explosive growth of the Internet, an increasing number of databases are becoming Web accessible through form-based search interfaces, and many of these sources are database-driven E-commerce sites. It is of great importance to provide users with a unified access to multiple E-commerce search engines (ESEs) selling similar products (services can also be treated as some kind of products) because this will allow users to search and compare products from multiple sites with ease. In this paper, we call a system that supports the unified access to multiple ESEs as an E-commerce metasearch engine (EMSE for short). Currently, a number of EMSEs exist on the Internet, such as www.addall.com, www.mysimon.com, www.pricegrabber.com, and www.shopping.com. However, the techniques used to create them are

not publicly reported. To the best of our knowledge, existing EMSEs are built either manually or semi-automatically. Furthermore, as ESEs operate autonomously, changes/upgrades to them may affect the operation of an EMSE. As a result, maintaining the operation of an EMSE is a costly long-term effort.

Our E-Metabase project aims to *maximally* automate the process of building large-scale EMSEs so as to significantly reduce the cost of building and maintaining EMSEs. This project consists of a number of components. First, a special crawler crawls the Web and identifies ESEs from the fetched Web pages. Second, an interface clustering component clusters the found ESEs into different groups such that the ESEs in the same group sell the same type of products (i.e., these ESEs are in the same domain). Third, an interface integrator integrates the interfaces of the ESEs in each group into a unified interface, which becomes the interface of the EMSE for this group. Fourth, a query mapper maps global queries specified on the unified interface to queries specific to the underlying ESE interfaces. Fifth, for each ESE, an ESE connector passes queries to and receives the results from the ESE. Sixth, for each ESE, a result extractor extracts the product information from the result pages returned by the ESE. Finally, a result merger combines the extracted results from different ESEs into a single list for presentation to the user. In this paper, we focus on the interface integration step of the E-Metabase project.

WISE-Integrator is designed to automate the interface integration step. It concentrates on the HTML form-based Web search interfaces like that in Figure 1. Such search interfaces have the following characteristics: (1) they are contained in HTML pages and are often supported by back-end databases storing the product information; (2) they often reflect (a portion of) the schema of the underlying product; (3) they allow users to specify more precise queries than using keywords alone. For a given set of Web search interfaces of a domain, interface integration consists of the following two major tasks: (a) identify the matching attributes across the interfaces; and (b) construct the unified interface based on these matching attributes. To be fully automated, the information that is needed for interface integration must be extracted from the HTML pages of the interfaces automatically. Therefore, WISE-Integrator also has an interface extraction component. Note that WISE-Integrator is applied to each group of ESEs produced by the ESE clustering step separately. Hence, in this paper, without loss of generality, we assume that all ESEs under consideration are in the same domain.

This paper has the following contributions. First, we provide a comprehensive solution to the search interface integration problem. Our solution includes Web search interface extraction, interface schema matching, global attribute generation, unified search interface construction, and unified interface maintenance. In contrast, the related existing works (e.g., [DDH01, DR02, HC03, MBR01]) deal with only schema matching. Second, our solution is fully automated using only general (i.e., domain-independent) knowledge while most existing works employ manual or semi-automatic techniques. A key issue in schema matching is to identify matching attributes from different interfaces. We explore a rich set of *meta-information* contained in Web search interfaces and propose a two-step clustering method to tackle this issue. Furthermore, this method also solves a rarely addressed issue, i.e., finding appropriate names for attributes in the unified interface automatically. Third, we present a technique to automatically extract information, from the HTML pages of the interfaces, that is needed to perform automatic interface integration. Our experimental results based on 143 real-world search interfaces in four different domains indicate that WISE-Integrator can achieve high attribute matching accuracy and can produce high-quality integrated search interfaces without human interactions.

The rest of this paper is organized as follows. In Section 2, we introduce the interface representation model used by WISE-Integrator. In Section 3, we discuss how to automatically construct the interface representation model. In Section 4, we present our method for identifying matching attributes. In Section 5, we discuss the generation of the global attributes, including merging attribute domains. In Section 6, we discuss the unified interface construction. In Section 7, we describe the maintenance of the unified interface. In Section 8, we report the experimental results and briefly describe our operational prototype system WISE-Integrator. In Section 9, we review previous research works related to our work. Finally, in Section 10, we provide some concluding remarks.

2 Interface representation

A Web search interface for e-commerce is usually presented through an HTML **form** in a Web page [HTL4]. It typically contains several form control **elements** such as *textbox* (single-line text input), *radio button*, *checkbox* and *selection list* (pull-down menu) that allow users to enter search queries.

Unlike traditional database systems, Web search interfaces have no explicit definitions of *schema* and *meta-data*. The schema reflected on a search interface is embedded in an HTML page, and is composed of labels and elements that are laid out on the interface in such a way with only human users in mind. To enable automatic interface integration, useful information related to the schema of each search interface needs to be identified, extracted, and properly organized. Indeed, interface schemas contain much useful meta-information that is specific to Web search interfaces. For example, in Figure 2, attributes Publication date, Publication Year, Price Range and Author have their own special presentation and composition. And from the name and the composition of an attribute, we may learn what semantics it has for specifying a query condition and what type of value it may take semantically. Such special features of Web search interfaces provide rich information in aiding attribute matching. In this section, we consider how to represent the information on search interfaces useful for the purpose of *interface integration*. In Section 3, we will discuss the automatic extraction of search interfaces based on this representation.

Fig. 1. The book search interface of amazon.com

Fig. 2. Examples of element relationship type

Each ESE interface can be conceptually viewed as a partial export schema of the underlying product database. In our interface representation, each interface consists of an ordered list of **attributes** and each attribute has one or more

associated **elements**. For example, in Figure 1 the attribute Author has four associated elements including a textbox and three radio buttons, and the attribute Publication date has two elements. Each attribute usually has a **label** (descriptive text) associated with it, which becomes the **attribute name**. Each element has a **format** that is the input format of the element. There are generally four types of formats: *textbox*, *radio button*, *checkbox* and *selection list*. Each element also has a **domain** that defines the set of **values** that can be used to instantiate the element when forming a query. Such values are also called attribute values. Textbox allows users to input whatever value they want while a selection list usually has a list of values (options), and a radio button/checkbox usually has a single value (note that the text/label associated with a radio button/checkbox is treated as the value of the radio button/checkbox in this paper). Often multiple checkboxes or multiple radio buttons are used together to accomplish the same function as a selection list. Labels and elements of an attribute often also indicate the **domain type** of the attribute. Four domain types are currently defined and they are *range*, *finite* (with a finite number of values possible but no range semantics), *infinite* (with possibly unlimited number of values, e.g., textbox, but no range semantics) and *Boolean* (has a single checkbox used to mark a yes/no selection). For example, in Figure 2, Publication Year is of a range type because a range condition can be specified. Each element or a group of elements may have its **default value**, which is used to help forming a query when a user does not make a different selection. For each attribute, though its values are treated as alphabetic strings when sent to the server, the attribute still semantically has its **value type**. Seven value types are currently considered in our model and they are *date, time, datetime, currency, number*, *char* and *id*. The values of *datetime* contain both *date* and *time*. The *id* type indicates that its values are used for identification purpose (e.g., product number, ISBN). Whenever possible, the **unit** of the attribute values is also identified. For example, *kilogram* is a unit for *weight* while *kilometer* is a unit for *length* or *distance*. Finally, each attribute has its **layout position** on its local interface. The position value is determined by the layout order of the attributes on the interface. More important attributes are usually arranged ahead of less important ones.

In addition to the label of an attribute, each element of the attribute may have its own label when the attribute contains multiple elements. For example, in Figure 2, the attribute Publication Year has two textbox elements with their own labels "after" and "before", respectively. Element label helps define the semantic meaning of the element.

When an attribute has multiple associated elements, these elements are related in some way. We identify the following four types of relationships among related elements based on our observations.

- **Range type:** It refers to the situation where multiple elements are used to specify the range semantics of an attribute. For example, in Figure 2, the attribute Price Range has two related elements indicating the minimum and the maximum values allowed.
- **Part type:** It refers to the *part-of* relationship. For example, in Figure 2, the Author has two elements "First name" and "Last name", and each of them is part of Author. Range type may be considered as a special case of part type.
- **Group type:** Multiple checkboxes/radio buttons are sometimes used together to form a single semantic concept (attribute). In this case, the labels associated with the checkboxes/radio buttons are values of the attribute. In Figure 2, the attribute Platform has a group of checkboxes.
- **Constraint type:** An element can be used as a constraint for another element. For example, in Figure 2, the checkbox Exact phrase is used to specify whether or not the input words should be treated as an exact phrase. Without being related to the textbox the checkbox would be meaningless for the attribute. Another example is the

attribute Author in Figure 1, which has three radio buttons as its constraints. In both cases, the checkbox or radio buttons are called *constraint elements* while the texboxes are called *domain elements*.

To summarize, in our interface representation model, a Web search interface is represented as $F = (S, \{A_1, A_2, \ldots, A_n\})$, where S is the site specific information (e.g., product domain of the site, site URL, etc) and each A_i represents an attribute. Each attribute A_i is represented as $(L, P, DT, DF, VT, U, R_e, \{E_j, E_{j+1}, \ldots, E_k\}, C_a)$, where L is the *attribute label* (possibly empty) of A_i, P is the layout position of A_i, DT is the domain type of A_i, DF is the default value of A_i (possibly null and there is at most one default value for each group of checkboxes and radio buttons), VT is the value type of A_i, U is the unit of A_i (if applicable), $\{E_j, E_{j+1}, \ldots, E_k\}$ is a list of domain elements of A_i, R_e is the relationship type of the domain elements, and C_a identifies the list of constraint elements (possibly empty). If $j=k$ (i.e., the attribute has only one element), R_e is null. Each element E_m ($m=j...k$) is itself represented as (L_e, N, F_e, V, DV), where L_e is the *element label* (possibly empty), N is the *internal name* (there is an internal name for each element in an HTML form source code), F_e is the format (e.g., textbox, selection list, checkbox or radio button), V is the set of values of the element, and DV is the default value of the element (possibly null).

Example 2.1: For the search interface in Figure 2, the attribute Price Range can be represented as (Price Range, 4, *range*, *null*, *currency*, *USD*, *range*, {(*"Between US$"*, *"low"*, *textbox*, ∅, *null*), (*"And US$"*, *"high"*, *textbox*, ∅, *null*)},∅), where "low" and "high" are the internal names of the two textboxes in the HTML source code, and ∅ denotes an empty set.

3 Search interface extraction

In order to truly automate interface integration, the information in the interface representation model described in Section 2 needs to be obtained automatically. Two relevant works [KBG00, RGM01] have been reported in the literature (See Section 9 for more detail). In this section, we outline our technique used to implement WISE-*i*Extractor [HMYW03] – the search interface extraction tool we employed to aid the experiments reported in Section 8. This tool can be either used alone or embedded into WISE-Integrator as a sub-system. There are primarily two tasks in search interface extraction. First, given an ESE search interface, group logically related labels and elements such that each group represents a logical attribute of the underlying product. An attribute label should also be identified for each group. Second, extract the meta-information of attributes needed for interface integration. The first task is the most challenging problem in interface extraction due to the lack of fixed syntax or structure for organizing labels and elements.

3.1 Attribute extraction

We observe that labels and elements that represent the same attribute have certain layout relationships (e.g., they are usually close to each other), and that in most cases they have some information in common. The relative layout positions of labels and elements can be captured by an ***interface expression*** (**IEXP**). For a given search interface, its IEXP is a *string* consisting of three types of items, namely '*t*', '*e*' and '|', where '*t*' denotes a label/text, '*e*' denotes an element, and '|' denotes a row delimiter (i.e., a physical row border on the search interface). IEXP provides a high-level description of the layout of different labels and elements on the interface while ignoring the details like the values of the elements and

the actual implementations of row delimiters. As an example, the search interface in Figure 1 can be represented as "*te*|*eee*|*te*|*eee*|*te*|*eee*|*te*|*te*|*t*|*te*|*te*|*te*|*te*|*tee*|*t*|*te*", where the first '*t*' denotes the label "Author", the first '*e*' denotes the textbox following the label "Author", the first '|' is the first row delimiter, and the following three '*e*'s denote the three radio buttons below the textbox (Note that the text associated with a radio button/checkbox is treated as the value of the element here; thus the text and its radio button/checkbox together are considered as a single entity).

We employ a two-step approach to automatically extract the attributes from the Web page containing a search interface. In the first step, the IEXP of the interface is constructed. This step is outlined below. We start from the tag "<form>" of the search engine form and continue until tag "</form>" is reached. When a label, an element or a row delimiter is encountered, we append a '*t*', an '*e*' or a '|' to the IEXP (it is empty initially) accordingly. The delimiter is identified by "<p>", "<br>" and "<tr>" tags in the HTML source code. In the second step, based on the IEXP, labels and elements are grouped such that each group corresponds to an attribute. A **l**ayout-**e**xpression based e**x**traction approach (**LEX**) is developed for this task. As shown above, the IEXP of an interface organizes texts/labels and elements into multiple rows. For each element *e* in a row, LEX attempts to find the text either in the same row or some rows above the current row that is most likely to be the attribute label for *e*. Some special features of search interfaces are utilized in this step to associate elements with labels. Some of these features are: (a) texts ending with a colon are likely to be attribute labels; (b) an element and its attribute label are likely to appear in the same row, otherwise they are likely aligned vertically, i.e., they have the same position index in their respective rows; (c) an element is likely to appear close to its attribute label (i.e., the difference between their positions in the interface expression is small). These features are aggregated together and the text with the highest score is selected as the associated label for *e*. At the end, all elements that are associated with the same label as well as the labels of these elements form a logical attribute.

3.2 Meta-information extraction

In our interface representation model, four types of meta-information for each attribute are defined and they are domain type, value type, default value and unit. Each type of meta-information needs to be automatically extracted. Deriving such meta-information is relatively straightforward as described below.

1. *Domain type*. Four domain types are defined and they are *range, finite*, *infinite*, and *Boolean*. The domain type of an attribute can be derived from its label, the associated element(s) and the relationship between the elements. If the element(s) of the attribute have the range semantics (i.e., used to specify a range condition), the domain type of the attribute is *range*. For example, keywords "less than" and pattern "(from)?[\s0-9]+(to|-)?[\s0-9]+" both have range semantics. If an attribute has a list of pre-defined values for users to select and it involves no range semantics, the domain of the attribute is of *finite* type. An attribute with just a single checkbox is considered to have a *Boolean* domain type. *Boolean* type may be considered as a special case of the finite type because an attribute of Boolean type takes two (*finite*) values conceptually. In our model, Boolean type is separated from the regular *finite* type as this separation is helpful in finding matching attributes. An attribute with an *infinite* domain type usually consists of *textbox*(es) with no range semantics.
2. *Value type*. Value types defined in our model include *date, time, datetime, currency, id, number* and *char*. To identify *date, time, datetime, currency* and *id*, we provide a thesaurus for each type, which contains commonly used

keywords and patterns related to the value type. If the labels (attribute label and element label) or element values contain relevant keywords and patterns, the attribute's value type is determined. A pattern can be defined as a regular expression. For example, keywords "date" or regular pattern "[0-1]?[0-9]/[0-3]?[0-9]/([0-9]{2}|[0-9]{4})" imply a *date* value type; keywords "morning" or pattern "[0-2]?[0-9]:[0-6]?[0-9](\s(A|P)M)?" imply a *time* value type; having both *date* and *time* implies a *datetime* type; "$" implies a *currency* value type; keywords "ISBN" and "i.d." imply an *id* value type. If an attribute does not belong to one of these five value types, then we check if the values of each of its elements are numeric. If they are, the value type is declared to be *number*; otherwise the value type is *char*.

3. *Default value*. Not every attribute has a default value. If an attribute just contains textboxes, then the attribute has no default value. The default value may occur in a selection list, a group of radio buttons or a group of checkboxes. It is always marked as "checked" or "selected" in the HTML source code of a form. Therefore, it is easy to identify default values.
4. *Unit*. To identify the unit of an attribute, we construct a unit library that contains the most popular units in e-commerce, such as those for "currency", "weight", "age" and "date". From the labels and values of an attribute, we may get some information about its unit. Then we use the information and the library to derive the appropriate unit for the attribute. For example, if a label contains "US$", the implied unit is "USD"; "publication year" implies that the unit is "year". We may utilize other information such as the site URL to help determine the unit. For example, for a "price" attribute, if the site URL ends with "ca" or "uk" before the first "/" (not counting http://), the unit of the price would most likely be Canadian dollar ("CAD") or United Kingdom pound ("GBP").

In addition, if an attribute has multiple elements, we also identify their relationship automatically. In Section 2, we introduced four types of relationships between elements: range type, part type, group type, and constraint type. Group type and constraint type are easy to identify based on their definitions, and range type is already addressed in the above. If an attribute has multiple elements but the relationship is not one of the other three types, then a part type relationship is assumed. Furthermore, some widely used patterns, for example, a date has three parts (day, month and year) and a person name has "first name" and "last name", are also utilized to identify part type relationships.

4 Identifying matching attributes

To integrate multiple Web search interfaces, the first step is to identify matching attributes across these interfaces. In this section, we present our attribute matching method based on the interface representation model described in Section 2.

4.1 Normalization

Since search interfaces are designed autonomously by independent designers, each interface has its own naming convention for attribute names and values. Consequently, the same concept may be named differently. For example, "Authors" and "by author" represent the same concept, but have different names; "any keywords" and "keywords" are also matching attributes. To increase the chance of match, attribute names and element values are first normalized as follows.

- Convert each name or value string to its lower case equivalence.
- Remove all content in parentheses, including parentheses. The content within parentheses often has no real meaning to the name or value, for example, "author name(e.g., susan)" and "title word(s)".
- Replace any non-alphanumeric character by a space character.
- Tokenize each name/value using space, replace abbreviation and acronym (if any) [MBR01] and use WordNet to get the base form of each token. For example, "max" is an abbreviation of "maximum" and "author" is the base form of "authors".
- Remove non-content words (e.g., "the" and "of") when a name or a value consists of multiple words.

4.2 Semantic relationships

Identifying semantic relationships between concepts or objects is very important in database schema integration and Web source integration. To facilitate attribute matching, we identify the following three types of semantic relationships between terms (attribute names or element values): *Synonymy*, *Hypernymy* and *Meronymy* [Mil95, BCV01, BBB02].

- *Synonymy*. Two terms T_1 and T_2 are synonyms if they have similar meanings.
- *Hypernymy*. Term T_1 is a *hypernym* of term T_2 if T_1 is more generic than T_2. For example, *tree* is a hypernym of *maple*.
- *Meronymy*. Term T_1 is a *meronym* of term T_2 if T_1 is a part of T_2. For example, *first name* is a meronym of *name*.

Given a term, we use WordNet [Mil95, WDNT] to get its synonyms, hypernyms and meronyms, if applicable. However, hypernymy and meronymy terms that can be found from WordNet are very limited. In WISE-Integrator, we also identify hypernymy and meronymy relationships of two terms using the information in the interface representations. For example, suppose we have two interfaces, one has an attribute hardcover and the other has an attribute format that has a value "hardcover". From this, we can identify *format* as a hypernym of *hardcover*. Also, the *part* relationships of elements may be used to discover meronyms. For example, from a search interface that contains an attribute author with two parts, "first name" and "last name", we establish both *first name* and *last name* as meronyms of *author*.

4.3 Matching attributes

The problem of attribute matching is to identify the attributes from different schemas that semantically represent the same concept. This problem has been studied extensively for many years (e.g., [DDH01, DR02, HC03, LC00, MBR01]) but only recently has the attention been focused on the development of automated solutions. Inspired by the SEMINT approach in [LC00], we propose an automated solution to the problem of search interface attribute matching. Our solution explores the meta-information of attributes as described in our search interface representation model. To our knowledge, exploring the specific meta-information from search interfaces for automatic attribute matching in the context of search interface integration has rarely been studied.

SEMINT utilizes and extends the metadata characteristics in [LNE89] to determine matching attributes. SEMINT introduces three levels of metadata that can be used: attribute names (the dictionary level), field specification (the schema level, e.g., data type and primary key), and attribute values and patterns (data content level). However, SEMINT just focuses on using the metadata at the schema level and data content level to determine attribute correspondences. It

describes 20 characteristics at the two levels, such as data length, data type, nullable, primary key, default scale, minimum, maximum, average and so on. We adopt the basic idea of the SEMINT approach for the attribute-matching task in the sense that we also use metadata characteristics in multiple levels. Our approach differs from the SEMINT approach in four aspects. First, the set of characteristics used is different. For example, primary key information and maximum value are readily available in a database context but they are not available for interface integration. On the other hand, information such as element format applies to only interface integration. Second, we utilize all three levels of metadata instead of just two. Third, we classify matches based on different metadata into positive matches and predictive matches (see below). Fourth, SEMINT uses neural network techniques but we don't. Furthermore, the SEMINT approach does not address how to determine a global name for the group of matching attributes.

As mentioned above, in our approach, we use the three levels of metadata to determine matching attributes. At the dictionary level, we explore six types of possible matches on attribute names: exact match, approximate string match [WM92], Cosine similarity of names [FB92] (see Section 4.3.2), synonymy match, hypernymy match and meronymy match. At the schema level, domain type, value type, unit and default value are used. At the data content level, element values are used.

In our approach, we classify the different matches into two types: *positive* matches and *predictive* matches. *Positive* matches include exact name match, semantic (synonymy, hypernymy and meronymy) matches and value-based match. For the value-based match, we employ exact match, approximate string match, synonymy match and hypernymy match. When *enough* values from the two attributes are matched (a threshold is used), value-based match is recognized as succeeded. When one of the positive matches occurs, the corresponding attributes are recognized as matched. *Predictive* matches consist of approximate name match, Cosine similarity of names, and matches based on domain type, value type, unit, default value, and value pattern. Predictive matches must be sufficiently strong (based on a weight threshold) for two attributes to be recognized as matched.

These two types of matches, i.e., positive matches and predictive matches, are carried out through two clustering steps to be presented in the next two subsections.

4.3.1 Positive match based clustering

This is to group attributes into clusters based on the positive matches between attributes. *All* input interfaces are considered at the same time. There are three steps for the positive match based clustering.

- Group attributes into clusters based on the exact match of attribute names in all interfaces. After this step, all attributes in the same cluster have the same name. For each distinct attribute name, the number of interfaces having it is counted. Then the values of all the attributes in each cluster, if any, are unioned. Note that in general the same name may have different meanings in different interfaces, namely, there may be *homonyms*. However, since we consider only interfaces in the same domain, homonyms rarely occur.
- Merge the clusters produced in the first step into larger clusters based on the matching of attribute values and the semantic (synonymy, hypernymy and meronymy) matches of attribute names between clusters.
- Determine the **r**epresentative **a**ttribute **n**ame (**RAN**) for each cluster produced in the second step. This attribute name will be a candidate of the global attribute name to which other attributes in the cluster will be mapped. To

determine the RAN of a cluster, we employ a method based on the *generality rule* and the *majority rule* as described below. First, we build *hypernymy hierarchies* based on the attribute names in the cluster. The roots of these hierarchies represent the most general terms among the attribute names in the cluster. Next, we select the attribute name among the roots that appears in most interfaces in the cluster as the RAN for the cluster. As an example, consider a cluster containing four different attribute names: “format”, “binding type”, “hardcover” and “paperback”. Suppose two hypernymy hierarchies are generated, one has “binding type” as the parent of “hardcover” and “paperback”, and the other has “format” by itself. Then the RAN for this cluster will be chosen between “format” and “binding type” based on which of them appears in more interfaces.

In our approach, the positive match based clustering step performs the preliminary attribute matching and the RAN identification. This step just gleans the knowledge about what attributes should be matched based on the positive information. There are several reasons to perform this clustering. First, count the number of interfaces in which each attribute name appears; this information is important for determining the global attribute names. Second, determine the RAN of each cluster in advance. Third, make sure that attributes that should be matched (based on positive matches) are matched. This can simplify the comparisons in the predictive match based clustering step and reduce the possibility of mismatches. Our experiments indicate that this two-step clustering approach, i.e., performing positive match based clustering before predictive match based clustering (see next subsection), is effective.

Example 4.1: Consider the three interfaces, F1, F2 and F3 as shown in Table 1. After the positive match based clustering, the following eight clusters are generated:

{Title:F1, Title:F3}, RAN = Title;

{Title word:F2}, RAN = Title word;

{Author:F1, Author:F1, Author:F3}, RAN = Author;

{Format:F1, Binding type:F2, Format:F3}, RAN = Format;

{Publication date:F1}, RAN = Publication date;

{Publication year:F2}, RAN = Publication year;

{Release date:F3}, RAN = Release date;

{ISBN:F2}, RAN = ISBN;

Note that Format and Binding type are clustered together because they have matching values.

Interface	Attr. Name	DT	VT	U	DF	Values
F1	Title	infinite	char	null	null	null
	Author	infinite	char	null	null	null
	Format	finite	char	null	All formats	hardcover, paperback
	Publication date	range	date	year	null	null
F2	Title word	infinite	char	null	null	null
	Author	infinite	char	null	null	null
	ISBN	infinite	id	null	null	null
	Binding type	finite	char	null	All bindings	hardcover, paperback
	Publication year	infinite	date	year	null	null
F3	Title	infinite	char	null	null	null
	Author	infinite	char	null	null	null
	Format	finite	char	null	All formats	hardcover, softback,CD
	Release date	infinite	date	year	null	null

Table 1. Example interfaces for macthing attributes

4.3.2 Predictive match based clustering

The positive match based clustering step may fail to recognize some matching attributes. For example, attributes Publication year and Release date cannot be matched because do not have the same name or matching values. However, they may have similar meta-information such as similar domain type, value type and unit, which may help matching them. The predicative match based clustering is to explore such meta-information to help the identification of additional matching attributes.

In our previous work [HMY03], an intermediate unified interface is generated in the matching process. Every time when an attribute A_i from a local interface is considered, we compare it with all attributes in the intermediate unified interface to find the *best match*. When the matching score of the best match is above a threshold, attribute A_i is mapped to the corresponding global attribute, causing a possible re-generation of the global attribute. In this paper, we employ a different approach. We first cluster all potentially matching attributes together and then generate the global attribute for each group of matching attributes. To cluster attributes in this step, in our current implementation, we adopt and extend a single-pass clustering approach [SM83, YM98], and reconsider all local interfaces again. Our predicative match based clustering approach is described in detail below.

Initially, no cluster exists. When the first local interface is considered, each attribute A_i on the interface forms a cluster C_i by itself. Meanwhile, for each A_i, look up the results of the positive match based clustering to obtain the RAN of A_i (denoted RAN(A_i)), and add a mapping entry, i.e., RAN(A_i) $\rightarrow C_i$, to an *attribute-cluster thesaurus*. The attribute-cluster thesaurus is constructed incrementally during the matching process.

For each attribute A_i in the local interface considered next, the approach first looks up the attribute-cluster thesaurus to see if an attribute with the same name as A_i has already been mapped to an existing cluster. If yes, A_i is placed into the found cluster; otherwise, we look up the thesaurus to see if RAN(A_i) is mapped to an existing cluster. If a cluster is found, A_i is mapped to the cluster, and this mapping is added to the attribute-cluster thesaurus as an entry. Performing the above two lookups is to avoid the re-computation and to utilize past successful mappings. If both of the two lookups both fail, we compute the matching weight between A_i and each existing cluster C_k to determine which cluster should include A_i. The weight between A_i and C_k is the average of the weights between A_i and all the attributes in C_k, and the weight between two attributes is the sum of the their weights based on several predictive matching metrics to be introduced shortly.

After the weights between A_i and all clusters are computed, the cluster with the highest weight is selected (the tie is broken using heuristics such as selecting the cluster with more attributes). If this weight is greater than a threshold w, attribute A_i is mapped to the selected cluster; otherwise, we assume that no existing cluster is suitable for A_i and a new cluster is created based on A_i. In both cases, two mapping entries, one is the mapping from attribute A_i to the cluster and the other is from RAN(A_i) to the cluster, are added to the attribute-cluster thesaurus. The above process is continued until all interfaces to be integrated are processed.

We now present the metrics used to compute the weight between two attributes.

1) Approximate string match and substring match: An approximate string match of two attribute names is to find out if the *edit-distance* between the two name strings is within an allowed threshold T. We use the approximate string match

algorithm in [WM92] to carry out the match. If the edit-distance is within the allowed threshold, assign a positive weight *Wam*. In some cases, a single-word attribute name is a substring of some words in another attribute name and it is difficult to find an appropriate *T* to recognize such matches. Therefore, if the edit-distance is not within the allowed threshold, we also employ substring match between the two attributes. *Wam* is also assigned when such a match is recognized. If none of the above two matches is recognized, *Wam* is 0.

2) Cosine similarity of names: The Cosine similarity is widely used in the information retrieval field to measure the degree of similarity between two documents, or between a document and a query. Since attribute names may consist of multiple words, we use this approach to measure the similarity of two attribute names. The approach is also used in [Coh98]. We first tokenize each attribute name, get the term frequency of each term in each name string, and then apply the Cosine similarity function [SM83] to compute the similarity of the two strings.

3) Domain type match: If the two attributes have the same domain type, assign a weight *Wcd*; otherwise *Wcd* is 0. In addition, we observe that the *range* type is used much less often than other domain types. Thus, if both attributes have the *range* domain type, we double *Wcd*.

4) Value type match: As mentioned in Section 2, seven value types (*date, time, datetime, currency, number*, *char* and *id*) are considered in our approach. *Datetime* is considered to be compatible with both *date* and *time*. If the two attributes have the same or compatible value type, assign a weight *Wvtm*; otherwise *Wvtm* is 0.

5) Unit match: If the two attributes that have the same value type also have compatible units (e.g., US$ and CAN$ are compatible currency units because they are mutually convertible), assign a weight *Wcu*; otherwise, *Wcu* is 0.

6) Default value match: If the two attributes have matching default values or the two clusters containing the two attributes that are obtained in the positive match based clustering have some matching default values, assign a weight *Wdv*; otherwise *Wdv* is 0.

7) Value pattern match: For two numeric attributes, if the averages of all the values of the two attributes are close, assign a weight *Wvp*; otherwise *Wvp* is 0.

The final weight between two attributes A_i and A_j is the sum of the weights based on the above seven matching metrics (the optimal values of these weights are currently determined by experiments, see Section 8.2.2):

$$W(A_i, A_j) = Wam + Wvss + Wcd + Wvtm + Wcu + Wdv + Wvp$$

Example 4.2: Continuing with Example 4.1, suppose F1 is considered first, then the clusters formed by F1 is: C_1 = {Title:F1}, C_2 = {Author:F1}, C_3 = {Format:F1}, C_4 = {Publication date:F1}, and the attribute-cluster thesaurus = {Title→C_1, Author→C_2, Format→C_3, Publication date→C_4}. When F2 is considered, Title word and Publication year are clustered to C_1 and C_4, respectively, because they have the highest weights with the corresponding clusters and the weights are above the threshold; Author is clustered to C_2 because there is a corresponding entry in the attribute-cluster thesaurus; Binding type is clustered to C_3 because its RAN (i.e., Format) is in cluster C_3; ISBN forms a new cluster because its weight with any existing cluster is below the threshold. After F2 is considered the result is: C_1 = {Title:F1, Title word:F2}, C_2 = {Author:F1, Author:F2}, C_3 = {Format:F1, Binding type:F2}, C_4 = {Publication date:F1, Publication year:F2}, C_5 = {ISBN:F2}, and the attribute-cluster thesaurus = {Title→C_1, Title word→C_1, Author→C_2, Format→C_3, Binding type→C_3, Publication date→C_4, Publication year→C_4, ISBN→C_5}. After F3 is processed, the final result is: C_1 = {Title:F1, Title word:F2, Title:F3}, C_2 = {Author:F1, Author:F2, Author:F3}, C_3 = {Format:F1, Binding

type:F2, Format:F3}, C_4 = {Publication date:F1, Publication year:F2, Release date:F3}, C_5 = {ISBN:F2}, and the attribute-cluster thesaurus = {Title→C_1, Title word→C_1, Author→C_2, Format→C_3, Binding type→C_3, Publication date→C_4, Publication year→C_4, Release date→C_4, ISBN→C_5}.

5 Generating global attributes

After the two steps of clustering for matching attributes, each cluster semantically contains all the matching attributes from the input local search interfaces. Thus, each cluster corresponds to a global attribute in the unified interface. To generate the unified interface, we must first generate a global attribute for each cluster. The problem of generating a global attribute consists of the following three sub-problems:

1. *Determine the name of the global attribute*: The name of the global attribute should be derived from the names of the local attributes in the cluster to serve as their representative.
2. *Determine the domain type of the global attribute*: As mentioned previously, four attribute domain types are supported in our approach and they are *range*, *finite, infinite* and *Boolean*. A domain type that is compatible with the domain types of all the matching local attributes needs to be determined for the global attribute. The domain type determines the way the elements of the global attribute are presented in the unified interface, and how query conditions could be specified on the global attribute.
3. *Determine the values of the global attribute*: The values of the matching local attributes need to be merged to form the values of the global attribute. Merged values should be semantically unique and compatible with the local values. To facilitate this task, we consider attributes that take alphabetic values separately from those that take semantically numeric values.

In the following subsections, we discuss our solutions to the above three sub-problems in detail.

5.1 Determining global attribute name

As mentioned in section 4.3.1, after the positive match based clustering step, every local attribute has its corresponding RAN. A RAN is the most general term in the hypernymy hierarchy containing the term for the cluster. In our approach, after the predicative match based clustering step, we apply the *majority rule* to all the RANs in each cluster, and select the one that appears in most local interfaces as the global attribute name of the cluster. After the global attribute names are selected, an attribute-mapping table is constructed, which records mappings from every local attribute name to its corresponding global attribute name.

5.2 Determining global attribute domain type

It is possible for matching attributes to have different domain types. For example, in the book search interfaces used in our experiments, some Subject attributes have finite domains (i.e., they have pre-compiled values) while other Subject attributes have infinite domains (i.e, they allow users to enter a subject value). The domain type of the global attribute should maximally reconcile such differences. In our solution, we introduce a *hybrid* domain type for global attributes. *Hybrid* is the combination of *finite* and *infinite*. If an attribute has a hybrid domain type, users can either select from a list

of pre-compiled values or fill in a new value. Given a number of matching local attributes, we use the following rules to determine the domain type of the global attribute:

1) If all the matching attributes have the same domain type, then the global attribute also has this domain type.
2) If one of the matching attributes has a *range* type, then the domain type of the global attribute is also *range*.
3) If the matching attributes have mixed *finite*, *infinite*, *Boolean* and *hybrid* domain types with at least one of them being either *infinite* or *hybrid*, then the domain type of the global attribute is *hybrid*.
4) If the matching attributes have mixed *finite* and *Boolean* domain types, then the domain type of the global attribute is *finite*.

When the unified interface is constructed, the domain type of a global attribute is used to help determine the presentation format of the attribute on the unified interface. For example, the format of global attributes of infinite domain type will be implemented by textbox(es), and that of finite domain type by selection list(s).

5.3 Merging alphabetic domains

If some of the matching local attributes are of the *finite* domain type and have alphabetic values, we need to merge these values and form a value set for the global attribute. In WISE-Integrator, this is carried out in two phases. The first phase is during the positive match based clustering step discussed in Section 4.3.1. In this phase, due to the matching techniques employed (exact match, approximate string match, synonymy match and hypernymy match), semantic relationships between values in the same cluster are identified. In the second phase, we utilize the relationships between the values to merge them and generate a global value set.

Phase 2 consists of the following steps. First, we organize all the values into categories based on approximate string match, Cosine similarity match, synonymy match and hypernymy match. Thus, all values that are similar, synonymy or hypernymy are organized into the same category. Next, we solve the following two problems: (1) Which value should be chosen as the global value to represent similar and synonymy values in the same category? (2) How to provide global values to users if the values in the same category have hypernymy relationships? For the first problem, we keep a counter for each distinct local value and choose the most popular value among the similar and synonymy values as their global value. As to the second problem, we need to make a tradeoff between choosing the most **generic** concepts and choosing the most **specific** concepts because different choices will have different effects on *query cost* and *interface friendliness*. The cost of evaluating a global query includes the cost of invoking local ESEs to accept sub-queries, the cost of processing sub-queries at local ESEs, the cost of result transmission and the post-processing cost (e.g., result extraction and merging). If we choose only the most generic concepts as the global values and do not use the more specific ones, a query against the unified interface may need to be mapped to multiple values (corresponding to the more specific concepts) in some local interfaces, leading to multiple invocations to the local search engines, and thereby higher query evaluation cost. On the other hand, if we keep only the most specific concepts and ignore the more generic ones, users who want to query more generic concepts (i.e., have broader coverage) may have to submit multiple queries using the more specific concepts, resulting in less user-friendly interface. Our approach is to provide a *hierarchy* of values, including both the generic and the specific concepts, to users. Multiple categories may be formed for the values corresponding to each global attribute and a value hypernymy hierarchy is created for each category. Each hierarchy is

limited to at most three levels to make it easier to use. This approach remedies the problems of the previous two options and gives the users more flexibility to form their queries.

Example 5.1: Consider two Web bookstore interfaces, one has an attribute Subjects with values "Network", "Databases", "Programming languages" and so on, and the other has a matching attribute Subject with values "TCP/IP", "Wireless network", "Oracle", "Sybase", "Sql server", "C", "C++", "Java", "Pascal" and so on. After organizing the values into categories, the semantic relationships between the values from the two interfaces are identified. There are three possible ways to generate the global values. One is to use only the **generic** concept values, i.e., values from the first interface, namely "Network", "Databases", "Programming languages" etc. In this case, suppose a user wants to find information about Oracle. Since "Oracle" is not available, the user has to select "Databases" on the unified interface and submit the query. This global query will be mapped to three sub-queries for the second interface, namely "Oracle", "Sybase", and "Sql server". Obviously, searching based on "Sybase" and "Sql server" will waste the resources at the second site and return useless results to the user. The second option is to use only the **specific** concept values, i.e., the values from the second interface. In this case, a user who wants to find information about database (not of any specific type) needs to submit three queries using respectively "Oracle", "Sybase" and "Sql server". This is inconvenient to the user. Our approach will organize related values into a *hierarchy* (see the box on the right in Figure 3). In this case, if the user selects "Databases", the metasearch engine will generate three sub-queries for the second site on behalf of the user. On the other hand, if any of the three sub-concepts of "Databases" is selected, only that concept will be used for the second site but "Databases" will be used for the first site. This solution remedies the problems of the first two solutions.

We should point out that the values in a category sometimes do not form a hierarchy. In this case, we just provide a list of values without a hierarchy.

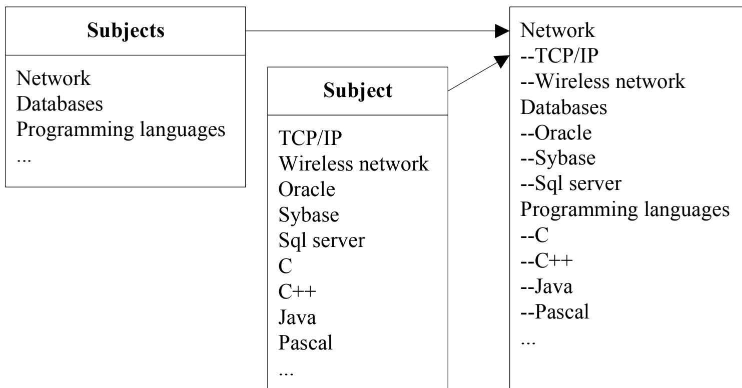

Fig. 3. An example of merging domain values

5.4 Merging numeric domains

To merge values of numeric domains, we need to perform the following tasks:

1) Resolve unit difference. In our approach, we build a unit relationship dictionary in advance for some popular units. The system can look up the dictionary to find out how to map one unit to another compatible unit. The numeric values in those attributes are transformed to the same global unit during value merging.

2) Understand the semantic differences involved.

3) Generate a global domain with query cost taken into consideration.

Fig. 4. Examples of different range formats

We identify two types of numeric domains: *range numeric domain* and *non-range numeric domain*. If the domains of the matching local attributes are *non-range numeric*, we just union all the values of these attributes for the global attribute.

For the rest of this subsection, we focus on attributes of *range numeric domain*. In Figure 4, we can see that there are various range formats. Two aspects need to be considered in resolving range differences, one is about *range modifiers* such as “from”, “to”, “less than”, “under” and so on, and the other is about *range width*. Figure 4 shows that different range domains may have different range modifiers and different range width. The basic resolution of range differences is to generate a global range domain that is compatible with the local range domains of the matching attributes.

For the *range numeric domain*, three types of formats can be identified as shown in Figure 4.

1) *One selection list*. The range type consists of only one selection list. The first four selection lists in Figure 4 are examples of this format.
2) *One selection list* and *one textbox*. This is exemplified by Publication date in Figure 4, which has two elements: a *selection list* for selecting a range modifier and a *textbox* for entering a numeric value.
3) *Two textboxes* or *two selection lists*. This type consists of two elements and each of them may be a *textbox* or a *selection list*. The examples are Price Range and Publication Year in Figure 4.

From Figure 4, we can see that ranges are often formed using numeric values and range modifiers together. To help the system understand different ranges, we need to let the system know the meanings of the range modifiers. For this purpose, we build a semantic dictionary that keeps commonly used range modifiers for numeric domains (see Table 2). In addition, we also save the meanings of other terms related to numeric values. For example, in Figure 4, we can see that “baby” and “teen” are in Reader age. We need to specify the meanings of these words for the system to understand them, for instance, “baby” may be interpreted as “under 3 years”, “teen” as “13-18 years”, and “adult” as “over 18 years”. This semantic dictionary can be used to build a range semantics table for each range attribute based on its values.

The range semantics table keeps multiple ranges corresponding to the original ranges in the element(s). As an example, consider the element that has the "less than" range modifier in Figure 4. From this element, we can obtain the following numeric values *5, 10, 15, 20, 25,* and *50*. We can also obtain phrases "all price ranges" and "less than". With the information and an approriate semantic dictionary for range modifiers, a range semantics table as shown in Table 3 can be built. Here the internal values are the values in the HTML text that correspond to the values of the element.

Range modifiers	**Meaning**
Less than	<
Over	>
Under	<
Greater than	>
From*	>=
To*	<=
Between*	>=
And*	<=
After	>
Before	<
…	…
All	All range
Any	All range

Table 2. Range modifiers (* modifiers to be used in pairs)

Lo	**Hi**	**Internal value**
0	5	'lessthan5'
0	10	'lessthan10'
0	15	'lessthan15'
0	20	'lessthan20'
0	25	'lessthan25'
0	50	'lessthan50'
0	∞	'allrange'

Table 3. A range semantics table

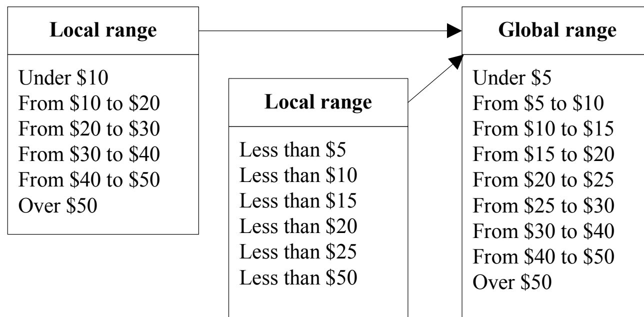

Fig. 5. An example of a global range domain

So far we have solved the first two problems of merging range numeric domains. The last thing we need to do is to generate global ranges that are compatible with the local ranges of the matching attributes. In general, a global query choosing a larger range on the unified interface will lead to more queries for some local sites and higher overall cost for evaluating the global query. Therefore, we should avoid having large ranges on the unified interface. In our current approach, we first obtain all the distinct values from the matching attributes, then sort them in ascending order, and finally generate a range using every two consecutive values. For the minimum and the maximum values, "under" and "over" range modifiers are used, respectively. A single selection list is used to implement each global range domain. This simple approach avoids generating large ranges on the unified interface and maps each global range to at most one local range for each local site. It, however, tends to produce too many small ranges on the unified interface when a large number of local interfaces are integrated. This is a problem we plan to investigate further in the future.

Example 5.2: Suppose in Figure 4 the two attributes with "from" and "less than" range modifiers are matched. The list of distinct numeric values under the two attributes is: 5, 10, 15, 20, 25, 30, 40, 50. From these values, the global range format can be easily obtained (see Figure 5).

6 Generating unified interface

After all the global attributes are generated, WISE-Integrator generates the unified interface in HTML format. Constructing the unified interface needs to consider three issues: (1) the presentation style of the attributes, (2) how to lay out the attributes on the interface (i.e,, attribute layout), and (3) which attributes to keep for the interface (i.e., attribute selection). The first issue has been addressed in Section 5.2. In this section, we address the remaining two issues.

6.1 Attribute layout

Intuitively, for an interface to be user-friendly, important and frequently used attributes should be laid out near the top of the interface. Each attribute has its layout position on its own local interface. These layout positions reflect the degrees of importance of the attributes as perceived by the local interface designers and their users. Usually, the first few attributes at the top of a local search interface are more frequently used than other attributes. In WISE-Integrator, for each global attribute, we aggregate the local positions of the corresponding local attributes into a global position, and arrange the global attributes in ascending order of their global positions. Specifically, the global layout position of a global attribute A_i is computed as follows:

$$P(A_i) = \sum_{j=1}^{m} P(A_i^j)$$

where $P(A_i)$ denotes the position of the i-th global attribute A_i, m is the number of local interfaces to be integrated, $P(A_i^j)$ is the layout position of the corresponding local attribute of A_i on the j-th local interface; $P(A_i^j)$ is assigned a value that is the total number of global attributes when no matching local attribute of A_i exists on the j-th local interface. Clearly, using this method, global attributes whose corresponding local attributes appear in high positions (the first position is the highest) on many local interfaces will likely appear in high positions on the unified interface, and global attributes whose local attributes rarely appear on local interfaces will likely be positioned near the bottom of the unified interface.

6.2 Attribute selection

When a large number of local interfaces are integrated, the unified interface may have too many attributes to be user-friendly. While some key attributes about the underlying product appear on most or all local interfaces, some less important attributes appear on only a small number of local interfaces. One way to address this problem is to trim some less important attributes from the unified interface. We use the global positions of global attributes to trim off less important attributes (those that have large global position values). A user-adjustable threshold for global position values can be used to control this. On the other hand, attribute trimming will reduce the capability of querying the unified interface. Therefore, we have to make a tradeoff between user-friendliness and the reduced querying capability.

7 Maintenance of the unified interface

Due to the dynamic nature of Web (i.e., Web sites may change/upgrade their Web search interfaces, new Web sites may appear, and some Web sites may be removed), it is necessary to deal with these changes in a timely manner to keep EMSEs operating properly. Hence, after a unified interface is generated, it is likely that some new local interfaces need to be added to or some existing local interfaces need to be removed from the unified interface from time to time. This requires maintaining the unified interface. WISE-Integrator is also designed to support this task.

For adding new local interfaces, the positive match based clustering step will incrementally cluster the attributes of these new local interfaces to the existing clusters that were established in the same step when building the previous unified interface. The RANs may need to be updated based on the current and previous statistical and semantic knowledge. Then, the predicative match based clustering is performed on these new local interfaces. Since a significant amount of mapping knowledge has been established when building the previous unified interface, clustering the attributes of these local interfaces will not take much additional time. When all the attributes of these local interfaces are mapped, some global attributes on the previous unified interface may need to be regenerated. The approach mentioned in Section 5 is still applicable for this purpose.

To remove a local interface from the unified interface, we remove the attribute names and their corresponding values from the clusters as well as the related mapping entries from the table that maps local attribute names to global attribute names. In addition, the entries corresponding to the affected attributes in the attribute-cluster thesaurus are also removed.

The addition of new interfaces and the removal of existing interfaces may also impact the domain types of the affected global attributes. The rules in Section 5.2 need to be applied again to determine the new domain type of each affected global attribute.

8 Experiments and implementation

In this section, we report the results of some experiments we carried out to evaluate the accuracies of our search interface extraction and interface integration techniques. We also briefly describe the implementation of WISE-Integrator.

To perform the experiments, we collected 143 real search interfaces from four domains: books (60), electronics (21), music (32) and movies (30). In general, the search interfaces of each domain differ significantly in attribute naming and composition. We constructed the interface representation for each search interface using the WISE-*i*Extractor component. Then we feed the interface representations of these search interfaces to the interface integration component to generate a unified search interface for each domain.

8.1 Evaluating the accuracy of interface extraction

WISE-*i*Extractor takes as input Web pages containing search interfaces. In some cases, search interfaces are ill-formatted, for example, missing some end tags. Such interfaces are tidied using *Jtidy* (http://lempinen.net/sami/jtidy)

before the extraction is performed. For each search interface, we manually identify the logical attributes on the interface, and then compare them with the results of WISE-*i*Extractor.

As discussed in Section 3, an attribute in general consists of up to three aspects of information: the name/label of the attribute, the set of the domain elements, and the set of the constraint elements (possibly empty). An attribute extracted by WISE-*i*Extractor is considered to match a manually extracted attribute only if they match on all of the three aspects. We call the match evaluation based on this requirement as an *attribute-level* evaluation. Our experimental results for the attribute-level evaluation are reported in Table 4, where "#Man" denotes the number of attributes that are identified manually, "#EXT" denotes the number of attributes that are correctly extracted by WISE-*i*Extractor. The overall accuracy is 94.62%. Generally it is more difficult to achieve high accuracy at the attribute level especially when attributes have multiple elements and when elements have their own labels. For example, suppose an attribute has three elements, but WISE-*i*Extractor only identified two of them; at the attribute level, this will be considered as a complete failure; but at the element level, the correctness rate will be 2/3 as two of the three elements are correctly identified.

Domain	**# Man**	**# EXT**	**Accuracy**	**Errors**		
				Label	Dom. elems	Const. elems
Books	370	351	**94.8%**	8	8	3
Electronics	146	137	**93.8%**	6	1	2
Movies	195	178	**91.3%**	11	1	5
Music	200	196	**98%**	2	2	0
Overall	911	862	**94.62%**	27	12	10

Table 4. The accuracy of attribute-level evaluation

To further evaluate the robustness of our approach, we carried out additional experiments using an independently collected dataset. The dataset is from DeLa [WL03] that was used for evaluating the label assignment algorithm in DeLa. This dataset has 27 search forms from three different domains (books, cars, and jobs). The overall attribute-level accuracy of our method on this dataset is 92.73%, which is about 2% lower than the accuracy of 94.62% obtained on our dataset. The main reason for this performance drop is that the DeLa dataset has more cases where an attribute has multiple elements arranged in different rows. To illustrate the problem such cases may cause, let us consider the attribute Other keywords in Figure 2: were the checkbox "Exact phase" below the textbox, it would be hard for LEX to know whether or not the checkbox is part of the attribute.

Domain	**# EXT**	**DT**	**VT**	**U**	**ER**	**DDC**
Books	351	347	349	341	347	343
Electronics	137	136	136	137	137	137
Movies	178	171	171	175	176	177
Music	196	191	194	194	192	196

Table 5. The accuracy of deriving attribute meta-information

WISE-*i*Extractor is also designed to obtain/derive meta-information, such as domain type, value type, unit, default value and the relationships between elements, for the extracted attributes. For each attribute with multiple elements, we also differentiate domain elements and constraint elements (this is recognized when the constraint type relationship between elements is identified). To evaluate the accuracy of obtaining attribute meta-information, we consider only those attributes that are correctly identified by WISE-*i*Extractor. Our experimental results for each domain are shown in Table 5, where "DT" denotes domain type, "VT" denotes value type, "U" denotes unit, "ER" denotes element relationship, and

"DDC" denotes the differentiation of domain elements and constraint elements. For example, for the book domain, out of the 351 attributes considered, the domain types of 347 attributes are correctly identified. The results show that nearly all needed meta-information can be correctly identified using our proposed algorithms.

8.2 Evaluating the accuracy of interface integration

8.2.1 Evaluation criteria

Three qualitative criteria for measuring the quality of a global conceptual schema in the context of database schema integration are proposed in [BLN86] and they are *Completeness and Correctness*, *Minimality* and *Understandability*. We rephrase these criteria and propose the following criteria to guide the evaluation of search interface integration.

Correctness. Attributes that should be matched are correctly and uniquely matched, and the domains of the matching attributes are correctly integrated. This criterion corresponds to the correctness and minimality measures in [BLN86].

Completeness. If a result can be retrieved directly via a local interface, then it can also be retrieved via the unified interface. In other words, query capabilities on any local search interface must be preserved on the unified interface. This criterion corresponds to the completeness measure in [BLN86].

Efficiency. The construction of a unified interface should consider query cost. While query cost is usually considered at the query evaluation time, a bad unified interface may incur high query cost despite of good query evaluation algorithms. For example, supporting only very wide range conditions on the unified interface may cause too many local queries to be submitted to a local search engine and too many unwanted results to be transmitted to the metasearch engine.

Friendliness. A unified interface should be simple and easy to understand and use by users. As an example, it is better to provide users a list of values for an attribute when these values are available for the attribute than letting users fill out the value without any knowledge. As another example, frequently used attributes should be arranged ahead of less frequently used ones. Although the friendliness is related to the understandability, the meanings of these two measures are not the same.

The completeness, efficiency and friendliness of the unified interface are taken into consideration by WISE-Integrator (see Sections 5 and 6). In the next subsection, we report our experimental results for completeness and correctness for matching attributes.

8.2.2 Experimental results

To evaluate attribute matching, we check how well local attributes are mapped to the global attributes in the unified interface. We consider the following three cases:

1) A local attribute is correctly mapped to a global attribute.
2) A local attribute is incorrectly mapped to a global attribute.
3) A local attribute is mapped to a global attribute *A*, but there is another global attribute *B* such that *A* and *B* are semantically the same and *B* represents more local attributes. This may occur when the attribute matching algorithm

described in Sectrion 4.3 fails to match the local attributes represented by *A* with those by *B*. In this case, the local attribute is considered to be incorrectly mapped. Note that if the local attribute is mapped to *B*, it is considered to be correctly mapped (i.e., this case is included in Case 1 above).

Basically, the *correctness* measure for attribute matching requires that local attributes that should be matched across all the input search interfaces be matched, and that local attributes that should not be matched not be matched. Our evaluation metric for correctness is called *attribute matching accuracy* (*ama*), which is the percentage of the correctly matched local attributes among all the local attributes. The formula we use for computing *ama* is:

$$ama = \frac{\sum_{i=1}^{n} m_i}{\sum_{i=1}^{n} a_i}$$

where n is the number of local interfaces used for integration, m_i is the number of the correctly mapped attributes on the i-th interface (*case 1*), and a_i is the number of attributes on the i-th interface.

The *completeness* measure requires that all the query capabilities on each local interface be preserved on the unified interface. Among the above three cases of attribute mapping, *case 2* will reduce the completeness because attributes in this case are incorrectly mapped to global attributes and using such global attributes may lead to incorrect results from some local search engines. However, *case 3* does not affect the retrieval of correct results because the semantics related to these attributes are correctly preserved on the unified interface. Therefore, *case 3* mappings do not reduce the completeness. We define the *attribute matching completeness* (*amc*) measure as follows:

$$amc = \frac{\sum_{i=1}^{n} (a_i - r_i)}{\sum_{i=1}^{n} a_i}$$

where r_i is the number of incorrectly mapped attributes on the i-th interface (*case 2*).

Domain	The number of Interfaces	Total Attributes	Case 1	Case 2	Case 3	ama(%)	amc(%)
Books	15 (1st round)	107	107	0	0	100	100
	30 (2nd round)	206	203	2	1	98.54	99.03
	45 (3rd round)	279	274	2	3	98.21	99.28
	60 (4th round)	370	357	8	5	96.49	97.83
Electronics	10 (1st round)	68	62	4	2	91.18	94.12
	21 (2nd round)	146	136	8	2	93.15	94.52
Music	10 (1st round)	65	61	2	2	93.85	96.92
	20 (2nd round)	120	109	4	7	90.8	96.67
	32 (3rd round)	200	186	5	9	93	97.5
Movies	10 (1st round)	69	65	0	4	94.2	100
	20 (2nd round)	137	131	1	5	95.62	99.27
	30 (3rd round)	193	187	1	5	96.89	99.48
Average		1960	1878	37	45	95.82	98.11

Table 6. The attribute matching correctness and completeness

We performed 4 rounds of experiments on book interfaces. In the first round, 15 interfaces were randomly selected and a unified interface was generated for them. In each subsequent round, 15 additional interfaces were randomly selected and added to the previously selected interfaces. Then a unified interface was generated for all the selected interfaces *from scratch*. We manually checked how well the attributes are matched. We also performed experiments using interfaces of the other three domains in a similar manner. The experimental results are shown in Table 6. On the average, the overall correctness and completeness of our approach for the four domains are *95.82*% and *98.11*%,

respectively. Furthermore, the results are remarkably stable (with all correctness and completeness values within a narrow range) despite the differences in the number of interfaces used and the product types. Our analysis indicates that some failures are caused by non-product-specific attributes, such as "sorting", "including product images" and "including product description", which are usually used to control how results are presented to users. If these attributes were not considered, better results would be obtained.

We also conducted experiments to evaluate the matching effectiveness of the predictive matching metrics used in the predictive match based clustering, and the impact of incremental integration of search interfaces.

Effectiveness of predictive matching metrics:

As discussed in Section 4.3, our approach to attribute matching across multiple interfaces first uses name matching and some value matching techniques to build the knowledge on what attribute should be positively matched. Then we use several meta-information based predictive matching metrics to identify additional matching attributes. Exploring and utilizing a rich set of attribute meta-information to perform attribute matching is an important feature of our approach. To find out the contribution of the predictive matching metrics toward the overall performance of our approach, we analyzed the results reported in Table 6. Specifically, for each correct match (i.e., *case 1* match), we check whether it is due to name and value based matching or due to predictive matching metrics based matching. The result of our analysis is shown in Table 7, where "#att" denotes the number of correctly matched attributes, "#pos" denotes the number of matched attributes due to name and value based matching, and "#pred" denotes the number of matched attributes due to predictive matching metrics based matching. Note that only the results when all the interfaces in each domain are used are reported in Table 7. The result indicates that overall about *17*% of all correct matches are due to the use of the predictive matching metrics. In summary, the use of the predictive matching metrics in our approach contributes significantly to the overall attribute matching correctness.

Domain	**# att**	**# pos**	**# pred**
Books	357	307	50
Electronics	136	118	18
Music	186	134	52
Movies	187	161	26
Overall	866	720	146

Table 7. The effectiveness of predictive matching metrics

Effectiveness of incremental integration:

Incremental integration allows new search interfaces to be integrated into an existing unified interface without starting the entire integration from scratch, thus allowing new unified interface to be generated faster. Supporting incremental integration of search interfaces is another feature of WISE-Integrator. Intuitively, considering all search interfaces at the same time is likely to lead to more accurate matching results than considering them incrementally. To see how well the attributes are matched incrementally, we used the same dataset and performed the same rounds of experiments as we did to obtain the results in Table 6, however, in each round here, the selected new interfaces are integrated into the unified interface generated in the prior round instead of starting from scratch. Table 8 shows the results of the incremental integration. Comparing to the results in Table 6, we can see that incremental integration degrades the matching accuracy only slightly.

In all experiments, the weights for the six metrics in Section 4.3.2 (the Cosine similarity match has no fixed weight) are: *Wam*=0.5, *Wcd*=0.1, *Wvtm*=0.4, *Wcu*=0.2, *Wdv*=0.6 and *Wvp*=0.1, and the weight threshold *w* is 0.62. These values are obtained from the experiments using the book interfaces and they are applied to the other interfaces without change. As the interfaces for books are very different from those for electronics, music and movies, the experimental results indicate that the above parameter/threshold values are robust.

Domain	The number of Interfaces	Total Attributes	Case 1	Case 2	Case 3	ama(%)	amc(%)
Books	15 (1st round)	107	107	0	0	100	100
	30 (2nd round)	206	203	2	1	98.54	99.03
	45 (3rd round)	279	274	3	2	98.21	98.92
	60 (4th round)	370	357	7	6	96.49	98.1
Electronics	10 (1st round)	68	62	4	2	91.18	94.12
	21 (2nd round)	146	136	7	3	93.15	95.2
Music	10 (1st round)	65	61	2	2	93.85	96.92
	20 (2nd round)	120	106	8	6	88.33	93.33
	32 (3rd round)	200	184	8	8	92	96
Movies	10 (1st round)	69	65	0	4	94.2	100
	20 (2nd round)	137	131	1	5	95.62	99.27
	30 (3rd round)	193	187	1	5	96.89	99.48
Average		1960	1873	43	44	95.56	97.80

Table 8. The attribute matching correctness and completeness for incremental integration

8.3 Prototype implementation

WISE-Integrator is developed using JDK1.4 and is now operational. To utilize WordNet, WordNet1.6 is embedded into the system through APIs based on the C language. The GUIs of the system are shown in Figure 6(a) and 6(b). The system has two components, one for search interface extraction and the other for interface integration. The search interface extraction component is implemented by WISE-*i*Extractor which can be used alone or embedded into WISE-Integrator as a sub-system.

WISE-*i*Extractor takes as input one or more HTML pages (or their URLs) containing search interfaces of ESEs. In general, a Web page might contain multiple search forms for different products (e.g., books, movies and music). Our system requires that all the search forms on a Web page be in the same product domain. After each input interface is extracted, the extractor shows its extracted search interface. For the interface in Figure 1, the extracted interface is shown in Figure 6(a) (only a fraction is shown). To make it easier to view the extracted logical attributes, they are separated by horizontal lines. The text displayed in each textbox (e.g., query-0 in the textbox next to “author”) is the internal name of the textbox. This allows users to check, based on the HTML source code of the form, if the textbox is grouped correctly with other elements and labels of the logical attribute. WISE-*i*Extractor then uses the extracted information of the search interface to construct the representation model in XML format to be used by the interface integration component.

The interface integration component of WISE-Integrator is to produce a unified search interface over the extracted search interfaces of the same domain. The system reads the interface representation of each ESE and displays the interface representation visually in a tree structure as shown in Figure 6(b). From the tree view, users can see all the extracted information for each local search interface. When the integration is completed, the unified interface and the attribute matching information are displayed. From Figure 6(b), we can see that the unified search interface (only a fraction is shown in this Figure) over multiple book search interfaces, and that the global attributes are arranged in the

order of their importance. A special feature of WISE-Integrator is that users can remove an existing interface from or add a new interface to the existing unified interface at any time on the fly, and WISE-Integrator will generate the new unified interface without starting from scratch (i.e., incremental maintenance is implemented). In addition, users can choose a parameter value to trim some less important attributes from the unified interface to make it more user-friendly.

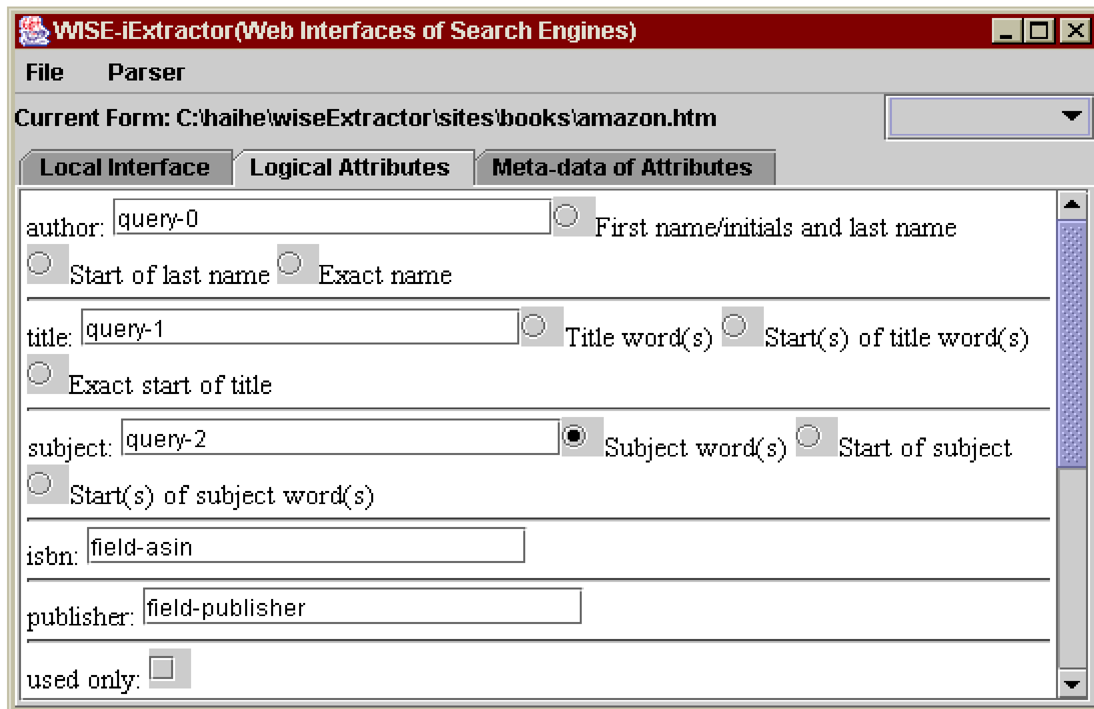

Fig. 6(a). A screen-shot of WISE-*i*Extractor

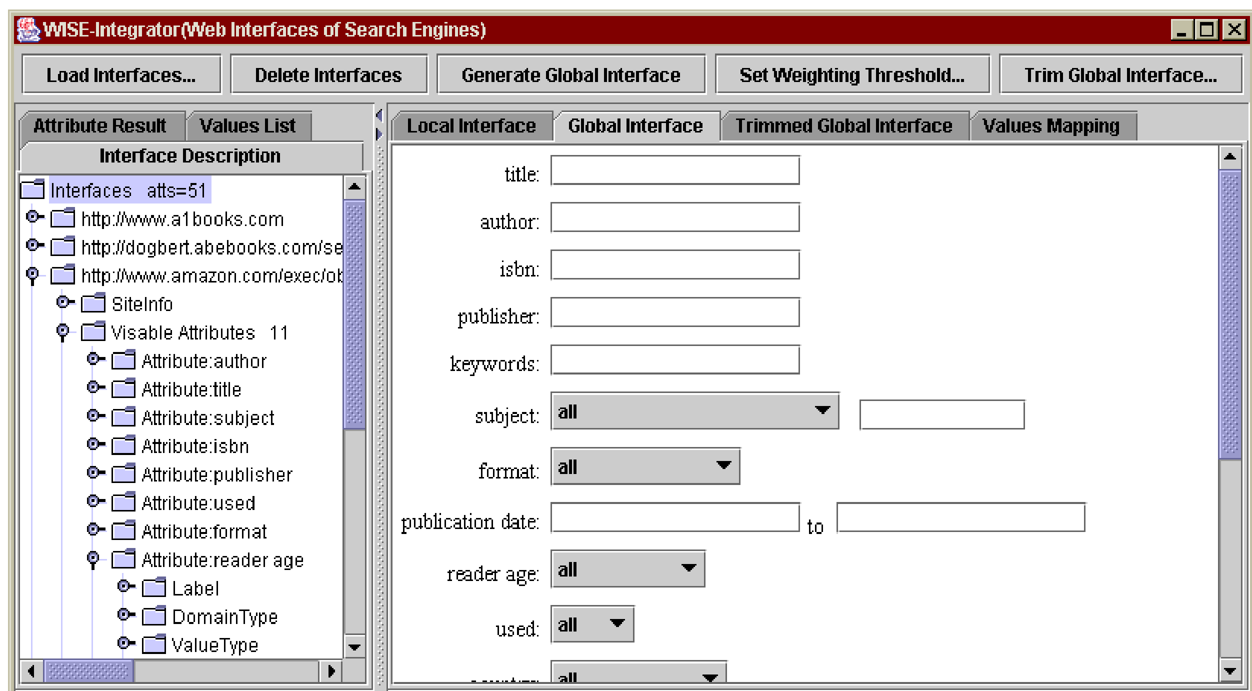

Fig. 6(b). A screen-shot of the interface integration component of WISE-Integrator

9 Related work

In this section, we compare our techniques with existing works related to Web search interface extraction and schema matching.

Web search interface extraction:

The works reported in [KBG00, RGM01] are closely related to search interface extraction. [KBG00] proposes a method useful for displaying and manipulating HTML forms on small PDA screens. The method breaks the entire HTML page into "chunks" and uses several heuristic algorithms to find a good matching label for each form element. The method LITE in [RGM01] identifies matching labels for form elements for the purpose of crawling the hidden web, but not for interface integration. It uses a layout engine to obtain candidate labels that are physically closest to an element in horizontal and vertical directions. The major differences between [KBG00, RGM01] and our LEX approach are as follows. First, the approaches in [KBG00, RGM01] are not completely *attribute-oriented*. In other words, they find labels for elements but not for attributes. If an attribute has multiple elements with their own (element) labels, the methods in [KBG00, RGM01] will probably fail to extract the attribute label. LEX aims to group all logically related labels and elements, and identify an appropriate attribute label for each group. Second, our solution is much more comprehensive than the solutions in [KBG00, RGM01]. We not only model the search interfaces and extract logical attributes, but also identify and extract a rich set of meta-information for extracted attributes. As a result, our approach is more suitable for search interface integration.

Schema matching:

A good survey of approaches for automatic schema matching can be found in [RB01]. [GKD97] predefines the mapping rules for each attribute and assembles these rules into a knowledge base for interpretation when a query is handled. [DEW96] predefines each domain description (including information about product attributes), and then uses some heuristics and mapping functions for the fields of each search interface but it provides little detail about user interface. [BBB02, BCV01] use description logics, common thesaurus and clustering techniques for semantic schema integration. It is a semi-automatic approach as the integration process still involves human interaction. Furthermore, the approach used for matching attributes is mainly based on name affinity and structure affinity, and only a few metadata (such as key and foreign key) of schemas are used. [LC00] uses neural network techniques and focuses on utilizing both schema level and data content level metadata to automatically identify matching attributes. Our approach has adopted some ideas from [LC00] but there are significant differences between the two approaches (see Section 4.3 for more comparison with [LC00]). [DDH01] uses and extends machine-learning techniques to semi-automatically find mappings between a source schema and the mediated schema. This approach needs human users to manually construct the semantic mappings between a small set of data sources and the mediated schema for training purpose. [MGR02] uses the idea of IP packet flooding to flood the similarity of elements. It converts each schema into a directed labeled graph. On the basis of the graph model, a part of the similarity of two elements propagates to their respective neighbors. No linguistic name matching is done beyond utilizing a simple string matcher to compare common prefixes and suffixes of literals. This approach is not suitable for search interfaces because name matching plays an important role in the integration of search interfaces. [DR02] discusses combining different matching algorithms in a flexible way and supports different ways to

combine match results. In [DR02], schemas are represented as rooted directed acyclic graphs. It maintains a matcher library of simple matchers such as approximate string matcher, synonym matcher, data type matcher and hybrid matchers (e.g., name matcher and structural matcher). It uses data type but not other schema or instance-level data to help find matches. Cupid [MBR01] investigates algorithms for generic schema matching. It combines a number of past techniques, such as linguistic-based matching and some metadata of schemas. It also proposes structure-matching algorithms for hierarchy schemas (tree structures) in which a structural similarity is computed between each pair of schema elements. However, the Cupid approach is not instance-based, i.e., attribute values are not used. Our experiences indicate that attribute values available on search interfaces are an important source of information for identifying matching attributes in search interface integration. The work reported in [HC03] is the most related to our work as it also focuses on Web search interfaces. [HC03] uses a statistical approach for schema integration of search interfaces of the *deep* web. It argues that as the Web sources proliferate, the aggregate schema vocabulary of the sources in the same domain tends to stabilize at a relatively small size, and that underlying these sources, there exists a unified *hidden schema model*. It uses only attribute names for statistical analysis, and it does not utilize other schema information such as domain type, value type and attribute values, which we find based on our experiments are important for accurate interface integration. Futhermore, [HC03] discusses only attribute matching, but not actual integration, for example, domain merging and global attribute name generation are not considered.

The main difference between our work and existing works is that we aim to perform comprehensive interface integration automatically, including attribute matching, value merging, format integration and the construction of an operational unified interface, while other existing works focus mostly on attribute matching, employing mostly manual or semi-automatic techniques. Furthermore, compared with other approaches for Web source integration, our approach utilizes a richer set of schema and instance level information to find matching attributes.

10 Concluding remarks

Providing integrated access to multiple data sources on the Web is an important aspect of information integration. The continued rapid increase of data sources on the Web calls for advanced tools that can automate the process of building integrated information retrieval systems. In this paper, in the context of creating tools to automate the process of building E-commerce metasearch engines, we presented our solution to the problem of automatically integrating the interfaces of E-commerce search engines into a unified search interface.

The problem of automatic search interface integration is significantly different from the schema integration problem involving traditional (relational) databases. Here we need to deal with not only schema integration, but also attribute value integration, format integration and layout integration. Furthermore, the schema information and important meta-information on search interfaces are not readily available and need to be automatically extracted to enable automatic search interface integration.

In this paper, we described in detail our techniques used to build WISE-Integrator – an automatic search interface integration tool. With appropriate representation of local search interfaces, WISE-Integrator automatically integrates them into a unified interface using only domain (application) independent knowledge. One of the key problems in automatic interface integration is to identify semantically matching attributes across multiple interfaces. We proposed a

two-step clustering approach based on positive matches and predictive matches to tackle this problem. This approach was shown to be highly effective based on our experimental results using 143 real search interfaces in four different domains. We also provided solutions to other important but rarely addressed issues in automatic interface integration such as attribute value integration, format integration and layout integration. In particular, our attribute value integration technique takes into consideration both the efficiency of the metasearch engine and the friendliness of the unified interface. During the integration, our approach considers all available input interfaces at the same time to maximally utilize the knowledge available on all of them. In this paper, we also presented the main ideas of our approach to automatically construct the interface representation of any E-commerce search engine from the Web page containing the search interface. Our search interface extraction tool, WISE-*i*Extractor, is capable of automatically grouping elements and labels into logical attributes and deriving a rich set of meta-information for each attribute.

For our future work, we plan to continue our research along several directions. First, having more hypernymy relationships can be useful for attribute matching and value merging. We plan to obtain additional hypernymy relationships from some online concept/topic hierarchies such as the Open Directory Hierarchy (http://dmoz.org) and the MeSH hierarchy (http://www.nlm.nih.gov/mesh/meshhome.html). Second, it is highly unlikely that an automatic, domain-independent and perfect attribute matching solution can be developed. Therefore, to provide a complete solution to the integration problem, it will be necessary to involve human integrators into the integration process. We plan to study how to most effectively involve human integrators and how to incorporate the involvement into WISE-Integrator. Third, we plan to look into how to extend our techniques for new applications. We believe that our approach, with appropriate extensions, can be applied to other application areas such as digital libraries and general Web databases.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work is supported in part by the following grants from National Science Foundation: IIS-0208574 and IIS-0208434.

References

[BLN86] C. Batini, M. Lenzerini, S. Navathe (1986) A Comparative Analysis of Methodologies for Database Schema Integration. ACM Computing Surveys 18(4):323-364

[BBB02] I. Benetti, D. Beneventano, S. Bergamaschi, F. Guerra and M. Vincini (2002) An Information Integration Framework for E-Commerce. IEEE Intelligent Systems 17(1):18-25

[BCV01] S. Bergamaschi, S. Castano, M. Vincini, D. Beneventano (2001) Semantic Integration of Heterogeneous Information Sources. Data and Knowledge Engineering 36(3):215-249

[Coh98] W. Cohen (1998) Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. In: Proc. 17th ACM SIGMOD International Conference on Management of Data, pp 201-212

[DR02] H. Do, E. Rahm (2002) COMA- A System for Flexible Combination of Schema Matching Approaches. In: Proc. 28th Internation Conference on Very Large Data Bases, pp 610-621

[DDH01] A. Doan, P. Domingos, A. Halevy (2001) Reconciling Schemas of Disparate Data Sources: A Machine-learning Approach. In: Proc. 20th ACM SIGMOD International Conference on Management of Data, pp 509-520

[DEW96] R. B.Doorenbos, O. Etzioni, and D. S.Weld (1996) A Scalable Comparision-Shopping Agent for the World Wide Web. Techinical Report UW-CSE-96-01-03, University of Washington.
[FB92] W. Frakes and R. Baeza-Yates (1992) Information Retrieval: Data Structures & Algorithms. Prentice Hall, Englewood Cliffs, New Jersey.
[GKD97] M. Genesereth, A. Keller, O. Duschka (1997) Infomaster: An Information Integration System. In: Proc. 16th ACM SIGMOD International Conference on Management of Data, pp 539-542
[HC03] B. He, K. Chang (2003) Statistical Schema Integration across Web Query Interfaces. In: Proc. 22nd ACM SIGMOD International Conference on Management of Data, pp 217-228
[HMY03] H. He, W. Meng, C. Yu and Z. Wu (2003) WISE-Integrator: An Automatic Integrator of Web Search Interfaces for E-commerce. In: Proc. 29th International Conference on Very Large Data Bases, pp 357-368
[HMYW03] H. He, W. Meng, C. Yu and Z. Wu (2003) WISE-iExtractor: Extracting and Modeling Web Search Interfaces for E-commerce Metasearch. Technical report, Computer Science, Binghamton University
[HTL4] HTML4: http://www.w3.org/TR/html4/
[KBG00] O. Kaljuvee, O. Buyukkokten, H. Garcia-Molina, and A. Paepcke (2000) Efficient Web Form Entry on PDAs. In: Proc. 10th International World Wide Web conference, pp 663-672
[LNE89] J. Larson, S. Navathe, R. Elmasri (1989) A Theory of Attribute Equivalence in Databases with Application to Schema Integration. IEEE Transactions on Software Engineering 15(4):449-463
[LC00] W. Li, and C. Clifton (2000) SEMINT: A Tool for Identifying Attribute Correspondences in Heterogeneous Databases Using Neural Networks. Data and Knowledge Engineering 33(1): 49-84
[MBR01] J. Madhavan, P. Bernstein, E. Rahm (2001) Generic Schema Matching with Cupid. In: Proc. 27th International Conference on Very Large Data Bases, pp 49-58
[MGR02] S. Melnik, H. Garcia-Molina, and E. Rahm (2002) Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching. In: Proc. 18th IEEE International Conference on Data Engineering, pp 117-128
[Mil95] A. Miller (1995) WordNet: A Lexical Database for English. Communications of the ACM 38(11): 39-41
[RGM01] S. Raghavan, H. Garcia-Molina (2001) Crawling the Hidden Web. In: Proc. 27th International Conference on Very Large Data Bases, pp 129-138
[RB01] E. Rahm, P. Bernstein (2001) A Survey of Approaches to Automatic Schema Matching. The VLDB Journal 10(4):334-350
[SM83] G. Salton and M. McGill (1983). Introduction to Moden Information Retrieval. McGraw-Hill, New York
[WDNT] WordNet: http://www.cogsci.princeton.edu
[WL03] J. Wang and F.H. Lochovsky (2003) Data Extraction and Label Assignment for Web Databases. In: Proc. 12nd International World Wide Web Conference, pp 187-196
[WM92] S. Wu and U. Manber (1992) Fast Text Searching Allowing Errors. Communications of the ACM 35(10):83-91
[YM98] C. Yu, and W. Meng (1998) Principles of Database Query Processing for Advanced Applications. Morgan Kaufmann, San Francisco