Integration of Web Information Sources by Mobile Users: Navigational Style of Integration and System Architecture

Wisut Sae-Tung, Tadashi OHMORI, and Mamoru HOSHI

The University of Electro-Communications, Tokyo 182-8585, Japan

Abstract. Recently, the number of Web sites suitable for mobile users has been rapidly increasing. Such Webs' contents, which may vary according to users' geographical locations, must be integrated into new simpler information sources for mobile users. How/when to do this integration is a problem, because different mobile users need different integrations according to their locations. A possible solution is to let each mobile user define, on his PDA, which pairs of Web sites should be integrated in what ways; later, this definition should be materialized by system-side servers. This paper describes a system supporting this requirement. For this goal, a new style of integration called *navigational integration* is proposed. Thereafter, our system architecture and a query language defining the integration are described.

1 Introduction

Currently, there are many Web sites providing suitable contents for mobile users. Typically, many of these Web sites (which are termed *Web Information Sources* or simply *WISs* in this paper) provide *location-dependent* contents, which vary according to users' current locations. Other WISs may provide the contents which are not location-dependent, but these contents are often closely-related with location-dependent contents. Mobile users must retrieve these Web-contents through their PDAs or smart phones, and must find useful information by combining these contents. To help such activities, a popular solution is that central managers of all WISs prepare *all* integrated views of all WISs [1–3]. However, this would be a heavy task in case of information services for mobile users, because different users will need different integrations according to the users' locations. Thus, it will be beneficial if each mobile user can freely integrate, on his PDA, one WIS and another related WIS into a new simpler WIS according to his location and requirement.

As an example, consider an exhibition of computer companies, and assume a WIS (denoted by WIS1) which announces, to each mobile user (= a participant), the list of 10 exhibitor-booths nearer to his current location. Assume also that a mobile user has already known another Web-document server (WIS2) which explains the booths of this exhibition. Then, consider that this user can integrate, on his PDA, the service of WIS1 with that of WIS2 into a new single WIS, in such a way that output-pages of WIS1 contain further links from the "booth" data-fields to related documents of WIS2. Then, this new WIS will become a useful portal site personalized for this user.

This paper describes a system architecture realizing this integration.

Figure 1 describes our assumptions of a system environment. There, an *area* is an autonomous organization of maintaining some WISs, whose contents are typically related with this area. (In default, an area corresponds to a geographical unit, such as a building or a university campus). A WIS is assumed to consist of *client applications* (= Web pages downloaded into users' browsers) and a backend

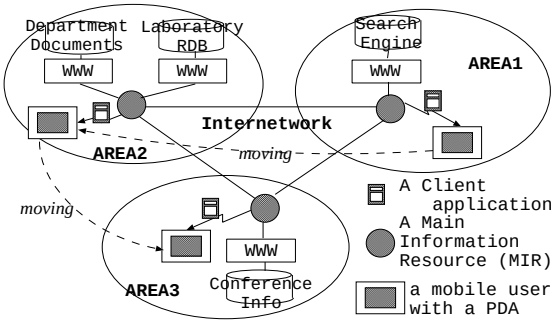


Fig. 1. A system environment

database server. Each area must have a *MIR*, which is a yellow page server announcing the existence of the WISs of this area. Access to some WISs can be restricted to that from certain areas. WISs may provide location-dependent contents (e.g., a map information nearer to users) or location-independent ones (e.g., a general search engine or a document server about its "owner" area).

Under these assumptions, our target is a situation where a user moves across different areas, finds useful WISs there, and wants to make an integrated information source from such WISs. While moving, a mobile user often disconnects his PDA from network. Thus, such an integration must be defined on users' PDAs without network connection. Thereafter, the definition should be materialized by remote system-servers when the network connection is set up again.

To realize these requirements, in the following, Section 2 describes a new style of integration called *navigational integration* for an easy integration style, and describes a system architecture for mobile users to do this integration by using only cached metadata of WISs. Section 3 describes a common data model wrapping WISs and a query command defining this integration. Section 4 describes how to create such a query with automatic semantic-conflict resolution on a user's PDA during network disconnection. Section 5 summarizes the paper.

2 Navigational Integration and System architecture

Navigational Integration: *Navigational integration*[4] is a style of integrating WISs by adding *derived links* between pages of the WISs. Intuitively, a *derived link* is a function-invoking link located at a user-specified data-field in a Web page of a given WIS, in order to access a new Web page of another WIS.

Figure 2 shows how two WISs work as an integrated resource by the navigational integration. The first WIS is assumed to have a client application that allows users to access the time- and location-dependent document information about one area (such as a conference program held in a building of a university). The second WIS is assumed to provide database information about a larger area (e.g., information about departments in a university) from a CGI-form input. Either of the WISs consists of a front page and a server function, where the front page accesses result pages through the server function. [1]

[1] The *front page* of a WIS refers to the 1st page returned when the WIS is accessed via its URL. *Result pages* refer to the other pages, which are generated by the server function of the WIS. We regard each of these Web pages as a *client application*.

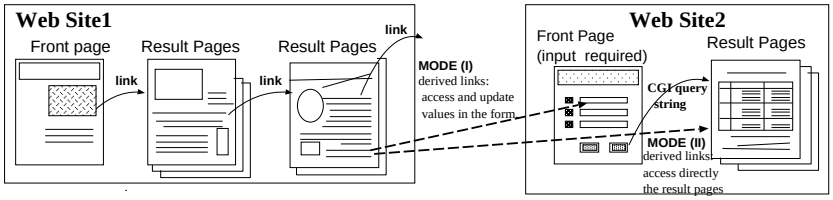


Fig. 2. Overview of navigational integration

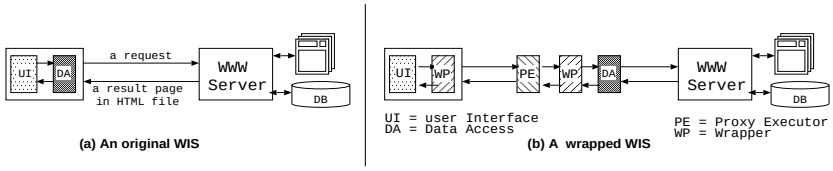


Fig. 3. An assumption of a WIS

Then, under these assumptions, the derived links depicted by dotted lines in Figure 2 are the links to pass data-values from result pages of the first WIS to the second WIS in such a way: (i) to set access-conditions of the front page of the second WIS and to allow a user to enter additional parameters for further operations; or (ii) to directly invoke the service of the second WIS and to access its result pages according to the passed data. (The passed data-values are those of the data-fields in which derived links are embedded). In this way, derived links are the links to *invoke service functions* of another WIS from one WIS.

Assumption of a WIS: We aim to let mobile users define, on their PDAs, the navigational integration. We use popular wrapper[2] approaches. Namely, WISs must be wrapped under a common data model; then we describe a navigational integration of given WISs **by a query command on this data model**. A user is requested to write such a query on his PDA, by using only cached data.

For the policy above, a WIS must consist of two parts: (i) a *client application*, which is the part downloaded into a user's browser, and (ii) a backend database server, which generates a result page and returns it. Furthermore, the client application must consist of two separate parts: a User Interface (UI) part and a Data Access (DA) part. (Figure 3-a shows this model). The UI part is a visible Web page to a user. The DA part is a call to the backend server under the input value from the UI part and to receive result pages from the server. (These constraints are satisfied by ordinary Web document servers and CGI-form based servers, and a certain class of Java Applets with backend servers). Then, for a given WIS, its UI part and its DA part must be wrapped separately by *wrappers* (Fig.3-b). Each of these wrapping descriptions is written in a common data model called *interface definitions*. Also, we prepare a *proxy executor* as another system-side server. As a result, when executing a navigational integration including a WIS of Fig.3-a, a proxy executor calls the wrappers of the UI and DA parts of this WIS individually, and lets them generate result-pages (modified to contain some derived links). Fig.3-b shows this behavior.

System Architecture: Figure 4 describes our system architecture. This figure is the case of three areas, where two areas maintain WISs individually. All areas share a common repository called *domain hierarchy*, which is used for automatically resolving semantic conflict of data between areas. Each area has

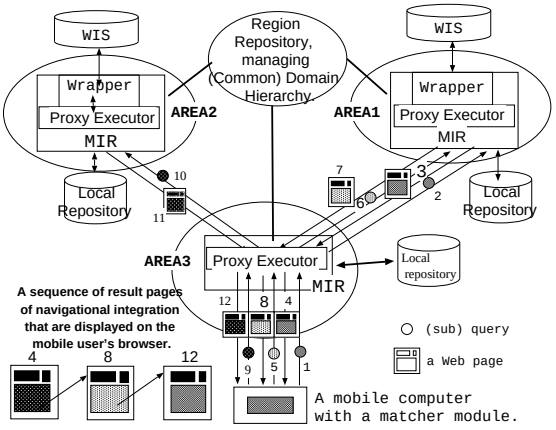


Fig. 4. A system architecture

a *main information resource* (MIR) (as a yellow-page server), which contains a wrapper and a proxy executor. The wrapper must wrap all WISs of this area; namely, it must maintain interface definitions of these WISs.

Under these preparations, our system of Fig.4 supports the following scenario under which a mobile user uses this system: Firstly, while a user's PDA is connected to the network, he is expected to find some interesting WISs about the area he exists now. Then, he must download only the interface definitions (IDs) of these WISs and a minimal set of *domain rules* about the IDs. Namely, only these metadata must be cached into his PDA (denoted by machineX here). (Note that the contents of WISs need not be cached). According to this manner, the user is considered to move across areas and collect IDs. Subsequently, the user disconnects machineX from the network, and activates a *matcher* on machineX. The *matcher* is a graphical query builder; under its assistance, the user is requested to build a query defining a navigational integration. (The matcher uses the downloaded IDs and domain rules so as to resolve semantic conflict among heterogeneous data). Finally, the user again sets up a connection to the network, and materializes the navigational integration by sending the query command to a proxy executor from machineX.

According to the scenario above, Figure 4 shows the steps (1-12) to materialize a query defining a navigational integration of two WISs. In this figure, a given query is firstly thrown to the nearest proxy executor (step 1). This executor then cooperates with the wrappers of the WISs and related proxy executors (step 2-3,6-7, and 10-11), materializes one Web page at a time according to the user's requests of link traversal, and replies back a new result-page by embedding new derived links according to the query command (step 4, 8 and 12).

3 Data Model, Path Expressions and Query Language

This section describes our common data model used by wrappers and a query language defining the navigational integration.

Data Model: Our data model is a minor variation on the object-oriented model designed for Web contents. To wrap a WIS, we regard its Web page as a set of complex objects. Thus, web pages of a common structure are regarded as

instances of a class. We describe the schema of this class by an *interface definition* (ID). Moreover, page-generating server-functions of the WIS are defined as methods of this interface definition. As a result, an ID has the following syntax:

```
database RepositoryName
address URLofProxyExecutor
class IDName
body
    attribute DataType [,attribute DataType]
method
public:
    MethodSignature [;MethodSignature]
private:
    MethodSignature [;MethodSignature]
implement ...
endclass
```

This syntax tells that a class *IDName* is maintained in a repository *RepositoryName* at a MIR of url *URLofProxyExecutor*. This ID has a list of (*attribute*, *data_type*) pairs, denoted by (A_1 T_1, ... , A_n T_n). An object (= a Web page of this class) has an instance value belonging to these ($attribute, data\_type$) pairs. The data type can be as follows:

1. *Domain* (DM). A *domain* is a pool of atomic data-values which have exact formats and meanings.
2. *listof* IDName. It represents a list of object-ids which are instances of IDName. This data type is used for describing a nested structure of Web pages.
3. *refto* IDName. It represents an object-id of IDName.
4. *link* IDName. It represents a link to another page wrapped by IDName.

We further use two operators ' |' (OR) and ',' (concatenation) on data types. These operators are combined with the above data types, so as to represent heterogeneous structures of Web data. For example, (DM_1| DM_2) represents a data type whose data-value may be in different formats. (DM_1, DM_2) describes a data type whose data-value is composed of several data-values. A *refto* (DM_1| $IDName_1$) tells a data type of different structures in one Web page. A *link* ($IDName_1$| $IDName_2$) represents a link to Web pages of different structures. (As a result, this model can represent DTDs).

Methods have a form $methodName(D_1$ AR_1 , ... , D_n $AR_n)$, where D_i is the domain of an argument AR_i. A public static method is used to generate pages of this class under given parameters. These methods can be used in query commands. The private methods are system-internal ones.

Example: Figure 5-a shows a WIS consisting of a CGI-form client with a database backend server. This server returns, when invoked, a list of presentations about a given topic; this list is ranked by both the floor numbers and the presentation times nearer to a user's location and time. Fig.5-b shows two IDs wrapping this WIS. The ID *BldGuideF* wraps the front page and its generator-function. The ID *FloorLabInfo* wraps the Data-Access part of this WIS. Namely, this ID represents a class of result-pages of this WIS. An instance of this ID consists of a name of laboratory, a floor number, and a list of objects of *Session*. The ID *FloorLabInfo* has one public static method *getByGuideConds*. Given arguments, this method retrieves result-pages from the wrapped native WIS. [2]

[2] Generally, an ID wrapping result-pages must have a pre-defined private method *getByStaticMethod*. This method is invoked when an original parent-page of these result-pages requests to generate result-pages through this ID. This request is issued either i) when a result page is accessed by its URL, or ii) when a double arrow (=>) in a path expression is evaluated. In either case, the request is once received by a proxy executor, and it in turn invokes *getByStaticMethod* of a corresponding wrapper; then, the wrapper in turn invokes an appropriate public static method of

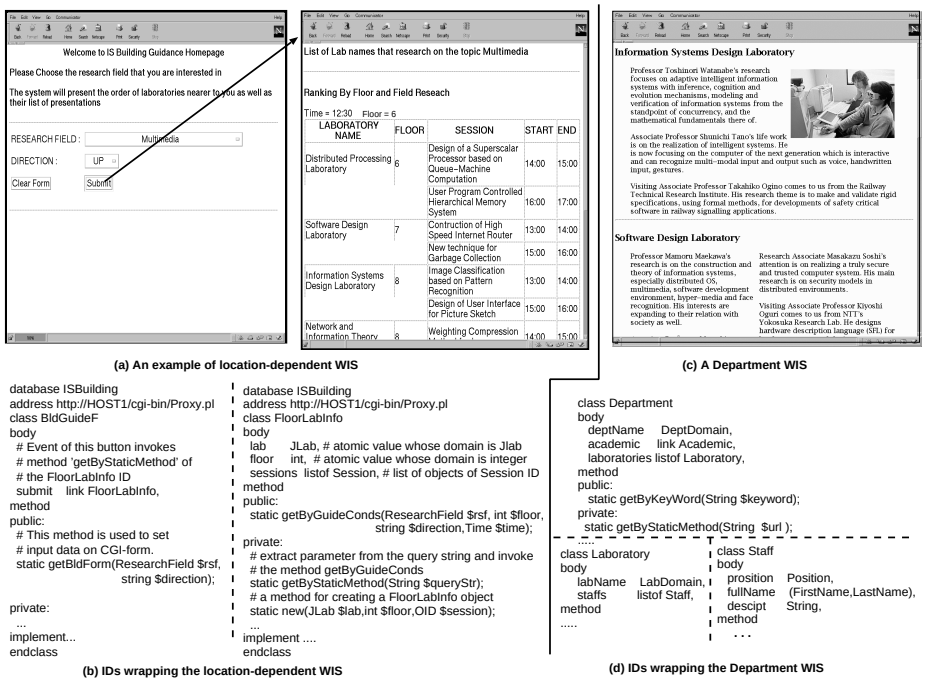


Fig. 5. Example of WISs and Interface Definitions

As another example, Figure 5-c shows a Web homepage of a department of a university. Figure 5-d shows IDs wrapping this Web page. They show how semistructured data are represented by our data model.[]

Path Expressions: A data-field on a Web page can be specified by a *path expression*. It has the form

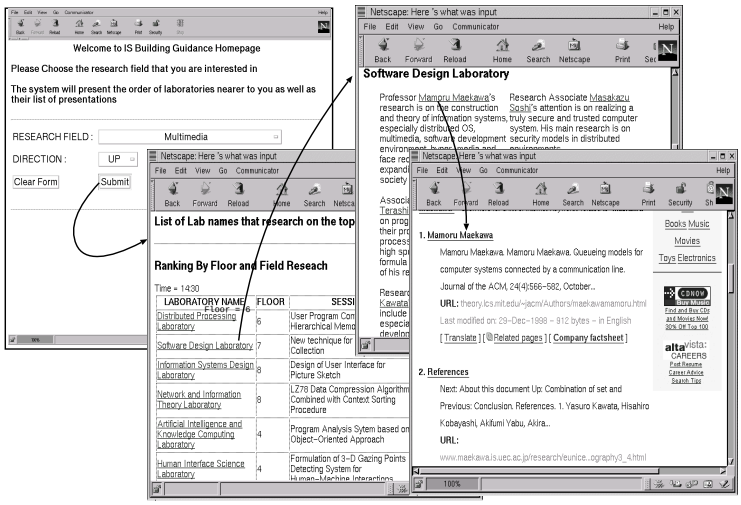
$$t_0 \rightarrow \{A_1\} \rightarrow ... \rightarrow \{A_i\} \Rightarrow IDName \rightarrow \{A_j\}[< Domain_j >] \rightarrow ... \rightarrow \{A_n\}$$

where t_0 is an object variable and A_i is an attribute. An arrow ($\rightarrow$) denotes traversing down an edge in a nested structure of a page. A double arrow ($\Rightarrow$) followed by an ID name that wraps its next result-page denotes traversing a link to a different Web page. When a data type of an attribute is defined by $'|'$ or $','$ operator, a symbol $< Domain >$ can be used to choose the data-value whose domain is $Domain$ in this attribute. (A symbol [] means an option).

As an example, assume that there is an index page containing links to some `Department` pages of Fig.5-c. Let `ISPage` be an interface definition of this index page, and let I be an instance object variable of `ISPage`. Then the following path expression

$$I \rightarrow \{depts\} \rightarrow \{dept\} \Rightarrow Department \rightarrow \{laboratories\} \rightarrow \{staffs\} \rightarrow \{fullName\} < FirstName >$$

this class so as to generate result pages. As an example, in Fig.5-b, the submit button of the front page is defined as an attribute whose data type is *link* to result pages. This 'submit' event is executed when a double arrow in a path expression $B \rightarrow$\{submit\}$\Rightarrow$.. is evaluated.




```
from   B in BldGuideF, D in Department, A in AltaSearch
source BldGuideF of ISBuilding on http://HOST1/cgi-bin/Proxy.pl
       Department of ISInformation on http://HOST2/cgi-bin/Proxy.pl
       AltaSearch of AltaVista on http://HOST3/cgi-bin/Proxy.pl
where BldGuideF->getBldForm('','')
and   Department->getByKeyWord(B->{submit}=>FloorLabInfo->{lab})
and   AltaSearch->getByKey(D->{laboratories}->{staffs}->{fullName})
```

Fig. 6. An example of navigational integration

is interpreted to get firstnames of all staffs in all *Department* pages in a given instance of ISPage.

Query Command: The navigational integration is defined by using a query command. It has a SQL-like syntax, as follows:

from OV_1 in ID_1, ... ,OV_n in ID_n
source ID_1 of DB_1 on PE_1,..., ID_n of DB_n on PE_n
where C_1 **and** ... **and** C_n

where: *(i)* OV_i is an object variable; *(ii)* ID_i is a name of an interface definition that wraps WIS_i; *(iii)* DB_i is a repository database name; *(iv)* PE_i is an url of a proxy executor; and *(v)* C_i is a condition to retrieve a Web page from WIS_i. The C_i is given by a form $ID_i \rightarrow methodName_i(arg_{(i-1)1}, ..., arg_{(i-1)j})$ where arguments $(arg_{(i-1)1}, ..., arg_{(i-1)j})$ are constants or path expressions specifying (atomic) data-values in the previous WIS (WIS_{i-1}). The source position of a derived link is set to the position of $arg_{(i-1)}x$, such that $arg_{(i-1)}x$ is the rightmost leaf at the deepest level in the set of the arguments $arg_{(i-1)1}$,...,$arg_{(i-1)j}$ when the Web page containing these arguments is regarded as a tree structure whose leaves are the attributes of atomic data-values. (The positions are determined by a page-specific wrapper. It wraps Web pages under a given interface definition, and transforms a page into a common data model).

Execution: To materialize navigational integration, a query command is sent to a proxy executor for execution. When a given query has a conditional expression C_1 **and**...**and** C_n, the proxy executor divides the execution into n units corresponding to C_i's. These are executed in the listed order. For each C_i, the proxy

executor again divides the C_i into some steps, by using the operator ' $\Rightarrow$' in C_i as a delimiter. The proxy executor executes these steps by one at a time. At the end of each step, the executor embeds derived links into the result page of the step, and returns the page to the user. (Refer to Figure 4. In implementation, a query is firstly stored in the proxy executor. Then, after each step, the derived links in a resulting page are embedded at the places specified by the path expressions of the remaining subquery. These links are written as CGI calls to the remaining subquery stored in the proxy executor).

Example: The query shown in Figure 6 describes a navigational integration of three WISs. The first WIS (WIS1) and the second WIS (WIS2) are the WISs already shown in Figure 5. The third WIS (WIS3) is the Altavista Search engine. Then, this figure also shows how this query is materialized. Let us explain how the materialization proceeds under this query. Firstly, the original front page of WIS1 is created through its wrapper by running BldGuideF(). Next, when a user clicks the submit button, the double arrow ($\Rightarrow$) of the argument of the second clause in the `where`-condition is evaluated, and thereby a result page of WIS1 is returned to the user. In this result page, derived links to invoke the server-function of WIS2 are now embedded at the laboratory name (`lab`) fields. When the usr clicks one of these links, the laboratory name is sent to invoke the method of the second `where`-clause. It then returns a department page (of WIS2) containing the required laboratory. In this page, new derived links are further embedded at the staffs' `fullName` fields; these links are to invoke the search engine (WIS3) under these `fullName` values. []

Generally, data-values of any entries used in navigational integration may be heterogeneous. In such a case, heterogeneous values/structures must be decomposed into atomic values by path expressions, and then the query-condition must be written in a general logical expression using the term C_i's. [3]

4 Domain Hierarchy for Resolving Semantic Conflict

This section describes *domain hierarchy* of Fig.4, and explains an algorithm with which a PDA caches a minimal set of metadata from the domain hierarchy. Thereafter, a conflict resolution algorithm on a PDA is described.

Domain Hierarchy: A *domain hierarchy* (DH) is a metadata repository for a given set of areas. A DH consists of multiple local domain hierarchies (LDHs) and a common domain hierarchy (CDH). Each LDH is maintained solely by one area (shown as the *local repository* in Figure 4). In contrast, a CDH is shared by all areas (shown as the *region repository* in Figure 4). Figure 7 shows an example of the domain hierarchy for two areas.

A LDH of an area must define, as its graph-nodes, all the domains that are locally used in this area. (These domains are those used as data types in the interface definitions of the WISs of this area). These definitions must describe the formats and meanings of these domains. A CDH must also define the domains which describe globally-standard data-formats and meanings, so as to exchange data among the areas.

In a given domain hierarchy, domains are linked to other domains. These links represent functions to do semantic conversion between the nodes of the links. By these links, domains are basically organized into a hierarchical structure. In this

[3] If an argument of a method requires a non-atomic (structured) value, we must give such a value by using `listof` constructors on path expressions; furthermore, the list structures between a passed value and a receiver's data type must be matched.

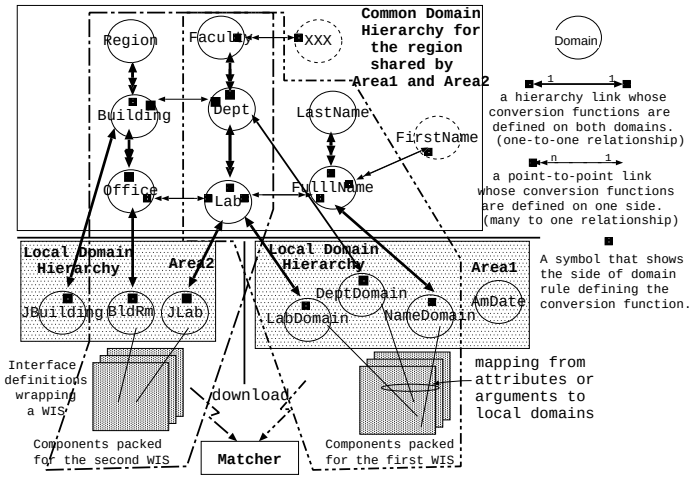


Fig. 7. Domain hierarchy and downloading mechanism

structure, a type of hierarchical relationship is a one-to-one relationship (data-format conversion), or an is-a relationship, or a part-of relationship. Therefore, for any given domain D, its parent domain is *equal to*, is *more general than*, or *contains (as a part of it) D*. The *hierarchy links* (depicted by bold links in Figure 7) represent such hierarchical relationships.

In addition to these hierarchical links, a given domain D may have other relationships with other domains D'_i $(i = 1, ..., n)$. We describe these (non-hierarchical) relationships by *point-to-point links* between D and D'_i. (These links are depicted by fine links in Figure 7). As a result, a domain D can have one hierarchy link to its parent node, and can have several point-to-point links to D'_i. (In implementation, each link is described by *conversion functions*. They convert data-semantics between the two associated domains).

Concerning the right to generate links, each LDH can freely link a domain in it to any domain in CDH or that in another LDH; in this case, it is enough that the description of this link is written only in the originating domain. (see the square symbols in Fig.7). In this way, a domain hierarchy as a whole must be maintained by distributed and partially-autonomous areas under a restriction of sharing CDH. In the following, by the term "*domain rules* of a domain D", we refer to the links which are directly described in D; thus, the domain rules of D are the semantic-conversion functions which D directly knows.

Matcher and Script Generation: As stated earlier, a minimal set of metadata must be cached into a mobile user's PDA. These metadata of a WIS are: (i) IDs of the WIS, (ii) domain-rules of the domains used in these IDs, (let us refer to these domains as X's), and (iii) the domain-rules of the ancestor domains of X. (An *ancestor domain* of a domain D is a domain reached by recursively-ascending from D along the hierarchy links).

Figure 7 shows this downloading process when a user requests to cache the metadata of a WIS. There, the domain rules and IDs (surrounded by the dotted line in Fig.7) of a WIS are packed as a package, and then it is downloaded and registered into the PDA.

After this caching, a mobile user calls a matcher on his PDA so as to build a query. The matcher then uses the downloaded domain-rules and interface definitions, and builds a sub-part of the domain hierarchy in it. Thereafter, when

the user tries to build a query on the PDA, he must pass attributes of one WIS to the method-arguments of another WIS. At this time, the matcher must resolve semantic conflict between the data of these WISs. For example, assume a request to resolve semantic conflict from data of a domain X to that of a domain Y. Then, according to the request, the matcher explores all the conversion paths between the two domains in the cached subpart of the domain hierarchy. Then, the matcher chooses the shortest path from X to Y to be a semantic conversion function from X to Y. As a result, a sequence of conversion functions along this path is used to resolve the conflict.

Example: Fig.7 shows a case where Area1 has the 1st WIS and Area2 has the 2nd WIS. Assume that the 1st WIS uses three domains (*LabDomain*, etc) in its result pages. Assume also that the 2nd WIS uses a domain *JLab* as an argument of its page-generating method. Then, a user must cache the metadata of these WISs (i.e., the packages surrounded by the dotted lines in this figure) into the PDA. Assume now that the user tries to build a navigational integration query which calls the server-method of the 2nd WIS from the data-fields of *LabDomain* in the 1st WIS's result pages. Then, the matcher must resolve conflict between *LabDomain* and *JLab*. Then, as far as the matcher knows, the shortest path from *LabDomain* to *JLab* is $LabDomain \rightarrow Lab \rightarrow JLab$. Thus, the matcher automatically inserts conversion functions associated with this path into the query. []

5 Concluding Remarks

This paper has proposed a new style of integration termed *navigational integration* of Web Information Sources (WISs), and has described a system architecture for mobile users to define this integration on their PDAs. We have described both a common data model wrapping WISs and a query language defining the navigational integration. A query defining the integration is later materialized by system-side servers. Also, we have described a *domain hierarchy* so as to support semantic-conflict resolution between WISs. We have explained how to cache a part of the domain hierarchy to users' PDAs and how to resolve semantic conflict when a query is built solely on a PDA.

Currently, a prototype system can deal with native WISs including Web-document servers, CGI-based database servers, and (Java-based) clickable map servers. We are currently examining domain hierarchy maintenance, effectiveness of the system for mobile users, and the inclusion of XML.

References

1. S. Chawathe. et al. The TSIMMIS project: Integration of heterogeneous information sources. In *Proc. 10th of Transactions of Information Processing Society of Japan Tech. Rep. DBS*, pp.7–18, Tokyo, Japan, 1994.
2. D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the world-wide web: Survey. *SIGMOD Record*, 3(27):59–74, September 1998.
3. P. M. P. Atzeni, G. Mecca. To weave the web. In *Proceeding of the 23th VLDB Conference*, pages 206–215, Athens, Greece, 1997.
4. W. Sae-Tung, T. Ohmori, and M. Hoshi. Navigational integration of autonomous web information sources by mobile users. *Proc. IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing*, pp.270–275, August 1999.