Regular Paper

An Information Integration Architecture for Mobile Users in WWW Environment

Wisut Sae-Tung,† Tadashi Ohmori† and Mamoru Hoshi†

Recent emerging technologies such as the WWW have significantly expanded the number of the services available to end users. Most of these services are provided through Web pages that include applications that act as clients of the information sources. We call them Web-embedded client applications. We propose a new information integration architecture and a set of tools that allow mobile users to integrate information provided through the Web-embedded client applications in the remote programming style. In our approach, Web-embedded client applications are wrapped to describe their services into abstract forms called interface definitions. The interface definitions can be seen as software components and loaded into a mobile unit. We provide an integrated building tool called mediator on mobile unit to explore information in the interface definitions, solve the schematic conflict problem and create customized applications in the disconnected state. We also present a preliminary experiment that indicates that our approach can be realized.

1. Introduction

1.1 Background and Motivation

Today's public networks such as the Internet contain a large number of information sources capable of providing specific services. These information sources provide their services through their Web pages that contain Web-embedded client applications. Here, a *Web-embedded client application* refers to the application that can be included in a Web page and acts as a client application to interact with the server system in an information source. It can be loaded with the Web page and executed by a browser. For example, the Web-embedded client applications may be simple static Web pages using a form-based CGI program or interactive Web pages using a Java applet [2] or Visual Basic [1] to access information of relational database server running at the information source.

In this environment, users carrying mobile computers will access information from these Web-embedded client applications that they find out within a current location while they are moving from one place to another. **Figure 1** displays the common operation that a mobile traveler finds out two Web-embedded client applications providing event and transportation information within an area he moves in. He finds out interesting events held in this area from the first source and how to go to that place from the second source. However, the user must select the output information of the first page, manipulate and transform it to the input conditions of the next page. From the point of view of users, this integration requires human participation. Moreover, schematic mismatch between information from different sources causes schema transformation required for integrating. This makes the integration process more complex. Furthermore, heterogeneity of Web-embedded client-applications implemented by different languages provide different interfaces for accessing information.

To solve these problems, we propose a new approach for mobile users to integrate the information provided through heterogeneous Web-

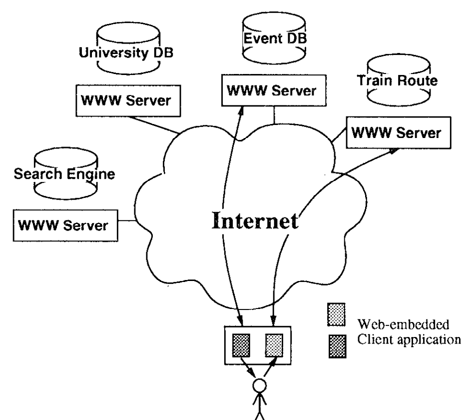


Fig. 1 A Common architecture of the WWW.

† Graduate School of Information Systems, The University of Electro-Communications

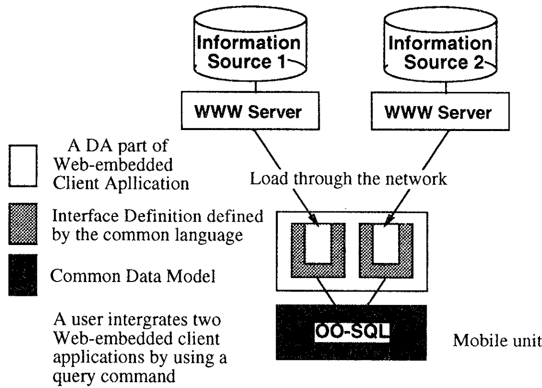


Fig. 2 Our information integration approach.

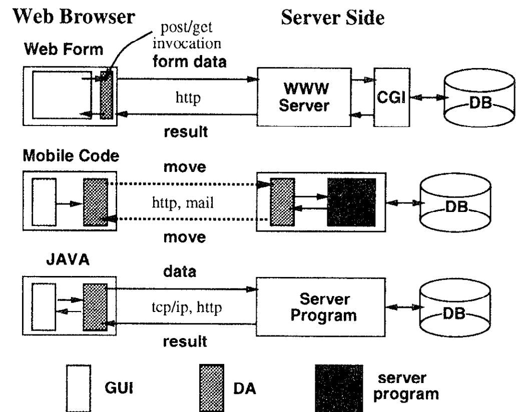


Fig. 3 A client application model.

embedded client applications. **Figure 2** shows the concept of information integration of our approach. Our proposal is to present the heterogeneous Web-embedded client applications in a common data model, integrate them using a query command in disconnected state and execute the query in the remote programming style. To realize this requirement, first, the Web-embedded client applications must be wrapped into the abstract forms of an object-relational model called *interface definitions* [6]. In addition, the interface definition also contains some rules that act as executable specifications to perform the specific tasks of the client application, such as the input validation, schematic mismatch resolution, etc. Second, the user loads and registers these interface definitions and their libraries into his mobile unit. Using the above abilities of the rules in the interface definitions, the user generates a query command (called OO-SQL command) in the disconnected state. This OO-SQL command is, on the mobile unit, converted into a script program. The script program is a program, expressed in a script language, that is able to communicate with the Web-embedded client applications and to access and integrate their information without schematic mismatch. Because of the low bandwidth connections through wireless communications, sending the script program to be executed on a server and receiving the result later is particularly well suited to this environment. Therefore, the user specifies his home server, sends the script program to a server capable of providing script execution and disconnects his mobile unit from the network. After the execution on that server is complete, the result is sent to be kept at his specified home server until the user reconnects the network to get it. As a result, no network connection is required while the user is generating the query command and waiting for the result of execution.

1.2 A Client Application Model

In general, the Web-embedded client application can be divided into two parts: *a GUI part* (Graphic User Interface) and *a DA part* (Data Access). The former part is responsible for gathering information from a user as input data while the latter part is responsible for passing the user's input as conditions to access information from information source. Because the script execution is done in disconnected state, the DA part that requires no interaction with a user during execution is the part wrapped into an interface definition. Therefore, the Web-embedded client applications used in our approach must have distinct separation between the GUI part and DA part. In the other words, any part of code of the DA part must not be enclosed into the GUI part. **Figure 3** shows examples of client types to which our approach can apply. The standardized CGI of the Web server can be invoked by wrapping the GET/POST method. Java applets and mobile code programs written by Aglet [8] or telescript [9] can also be wrapped if they have a distinct separation between two parts. In this way, the aim of our work that is different from the various integration projects is to integrate such various heterogeneous Web-embedded client applications.

1.3 The Model of the Information Integration Model

Based on the approach shown in Fig. 2, our model of the information integration can be shown in **Fig. 4**. This figure describes our

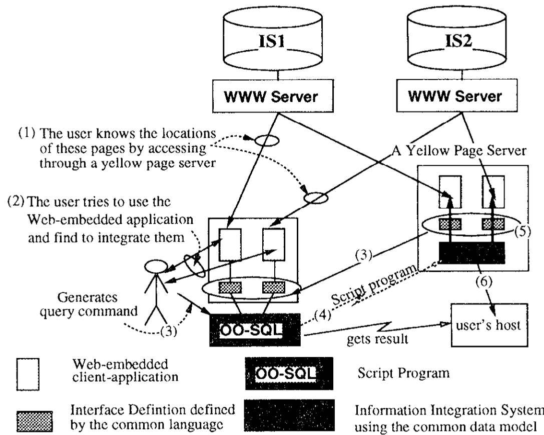


Fig. 4 An information integration model.

model by the following steps: In general, a user can know the locations of services by accessing a yellow page server that collects location information of services of interest (1). The interface definitions of these services are also managed by the yellow page server. After the user finds the desired Web-embedded client applications (2), he loads these interface definitions from the yellow page server (3). Because the interface definition contains the information adequate for generating a query command, the user can generate the query command and resolve a schematic conflict problem without connecting his mobile computer to the network. With the interface definitions, the user generates a query command, translates it to a script program, sends it to the yellow page server and disconnects his mobile unit from the network (4). At the yellow page server, it loads the Web-embedded client application from the network into its system. It converts the command in the script to invoke native methods of the Web-embedded client applications, executes and integrates the information based on the common data model (5). It sends the result back to a server specified by the user (6). The result is held at that server until the user reconnects to the network to get the result.

In this integration model, it illustrates many features and operations as follows:

1. Without disturbing the existing native code of Web-embedded client applications, the Web-embedded client applications are wrapped into interface definitions and provided to mobile users as components for describing their services.
2. Once the user loads the interface definitions, he can create his own customized applications by generating integrated views against the Web-embedded client applications by himself.
3. The user can resolve schematic conflict problem and integrate information in a disconnected manner.

This model of integration is suitable for mobile users who move around networks, collect interface definitions from various places and combine them to create their new "customized applications."

1.4 Comparison with Related Works

Various projects on the integration of heterogeneous sources are based on a common data model and a single query language. The TSIMMIS[4] project is a well-known pioneer system that uses this approach. However, it restricts end users to use only the static view defined in a middle layer called mediator[4], whereas our approach allows mobile users to define their own integrated views.

The distributed object-oriented frameworks such as CORBA[11] and DCOM[1] provide core object models, location transparency and programming language independence. In addition, Java Beans[2] and Active-X[1] are potential tools for "developers" to construct general client applications for the above distributed frameworks. These frameworks further need developers' effort to solve the schematic conflict problem. In contrast, our approach operates at a different level from their frameworks. We propose a new architecture for "mobile users" to create ad-hoc integration in the open environment by themselves. In the other words, those related works provide distributed environment and tools for developers to generate general applications while our uniqueness is to provide an much easier environment and tools for mobile users to generate customized applications from heterogeneous Web-embedded client applications in the open WWW environment.

This paper is organized as follows. Section 2 describes the system architecture of our approach. Section 3 describes the interface definition and a script generation. Section 4 describes a wrapper and its query execution. Section 5 describes preliminary experiment. Finally, we summarize the paper.

2. System Architecture

In this section, we describe major components in our model and the schematic conflict

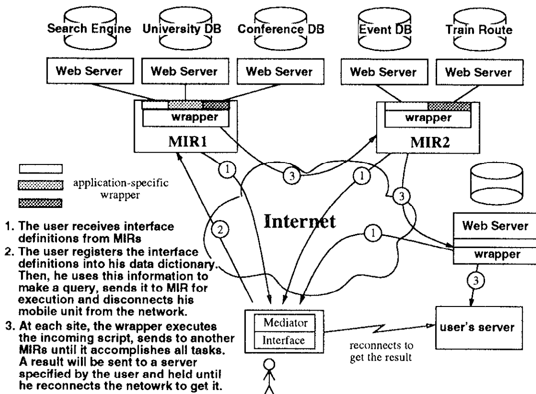


Fig. 5 A system architecture.

problems of integration.

2.1 Components in System Architecture

We now proceed to examine the information integration model in more detail. **Figure 5** shows the system architecture that implements the information integration model shown in Fig. 4. It consists of the components as follows:

Information Sources: Existing information sources shown as disk-shaped in Fig. 5 provide their information through their Web-embedded client applications.

To inform users about their service information, the client application and some necessary information for information integration must be wrapped in interface definition. We will describe the interface definition in detail later.

Mediator and User Interface: These two components are installed in a user's mobile computer and work together as a front-end system to generate script programs and present their result. We use object-oriented SQL (OO-SQL) and *Persistent Perl* [5] as a query language and a common language in our system respectively. The mediator communicates with the user through the user interface to help the user generate a query command. The mediator finally adds additional information written in the Persistent Perl into the user's OO-SQL command to form a script program and registers into the system for future use. With the user interface, the mediator sends the script program to remote servers for execution.

Wrapper: This module consists of a basic and application-specific integrated information service modules. With the interface definition, the wrapper converts the queries and additional information sent from the users into native commands of the client application, executes and converts the result represented in a common data model. The wrapper module does not necessarily reside in the same server as the information source. Most of the wrappers reside in a yellow page server. The wrapper in the yellow page server loads the Web-embedded client applications from original sites and communicates with them through its application-specific module for execution.

So far, there are important points we should note. From the point of view of users, integration is performed through the OO-SQL command. Because no global data schema is provided in our architecture, the process must accumulate knowledge useful for integration while it migrates between remote servers. Therefore, the mediator will include some additional knowledge written in the Persistent Perl together with the OO-SQL command to form a structured script program. This script program is used as a format transferred between the mediator and the wrapper, and also used between the wrapper and the wrapper. At each wrapper, the OO-SQL and information written in the Persistent Perl will be converted into native commands for execution.

MIR (Main Information Resource): A MIR acts as a yellow page server to assist mobile users in finding and integrating services of the information sources. It contains the wrappers and the interface definitions of all services of interest. It also provides links to the Web pages of each information source and their interface definitions.

2.2 Cell Design for Schematic Conflict Solution

Various projects on the integration of heterogeneous sources such as TSIMMIS and HERMES [7] focus on resolving the schematic conflict problem within the well-known underlying sources and leave the programming task for solving this problem to developers. This approach will work well on the static environment of centralized or distributed organization that is equivalent of a MIR in our approach. From the point of view of mobile computing, we can see that MIR is a *logical cell* that there is no conflict problem with its support range. However, our approach allows mobile users to decide by themselves to create views for integrating information from several Web-embedded client applications. We provide no global external view

at all, but rather provide interface rules in an interface definition to solve the schematic conflict problem while mobile users generate integration.

How can mobile users integrate information without the schematic conflict problem even though the range of integration covers different cells? Our solution is built on the variation the two projects described above. We use the following list of methods for designing a logical cell:

1. In general, even in different cells, a domain of each information has its standard formats for representation. Therefore, a cell designer set the standard format used in the cell and prepare the interface rule for handling the other standard formats to alleviate the conflict problems of the integration covers different cells. For example, the cell designer decides to use American format as the standard Date format and writes the interface rule for handling three formats; American, European and Japanese. This rule can be applied in a wide area made of different cells. This approach is adopted from that of the HERMES project.
2. In case that the first method can not solve the conflict problem, the interface rule for direct conversion between different domains of different cells should be prepared. This approach is adopted from that of the TSIMMIS project.
3. In case that the first and second methods can not solve the problem, the interface rule must provide users with "extract" functions, which are used to extract sub-components of information for making decision in integration. Such a sub-component can be used to perform partial matching integration.

We assume that a scale of MIR is not so large but the first and second methods listed above can hold in a much wider area made of different cells. The third method is an exceptional way we can use in case that the first and second ways can not be applied to solve the schematic conflict problem. **Figure 6** summarizes our strategies described above.

3. Interface Definitions and Script Generation

In this section we describe components in our system architecture in greater detail, provide an example of how Web-embedded client application is wrapped into an interface definition and how to integrate information by using interface definitions based on a query language.

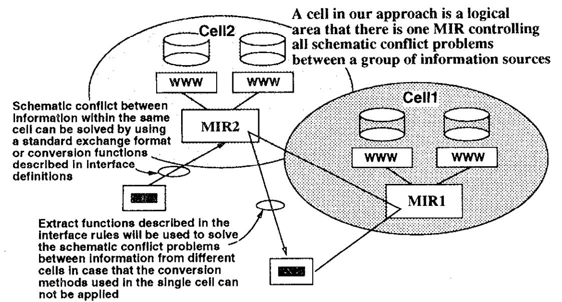


Fig. 6 A schematic conflict problem.

3.1 The Interface Definition

In this subsection, we will describe how to wrap Web-embedded client applications defined in Section 1.2. To wrap properties and behavior of these applications, We propose the Persistent Perl as a common language in our system. We also propose an interface definition to map the object-relational model into native information and tell a user what operations are available and how to invoke them. The general syntax of the interface definition is as follows:

database DatabaseName
address EMailAddress
class ClassName **inherit** SuperClass
body
    attribute Domain [,attribute Domain]
method
public:
    MethodSignature [,MethodSignature]
private:
    MethodSignature [,MethodSignature]
implement
    MethodImplementation
interface
    RuleDefinition [RuleDefinition]
endclass

where the above "RuleDefinition" takes the form:

Rule:
    Rule-Predicate
External:
    Function-Description
Default:
    Default-Value
Comment:
    Comment-String
Extract:
    Function-Description

To understand the syntax of the interface definition more clearly, consider a Web-embedded client application shown in **Fig. 7**. It is a Java

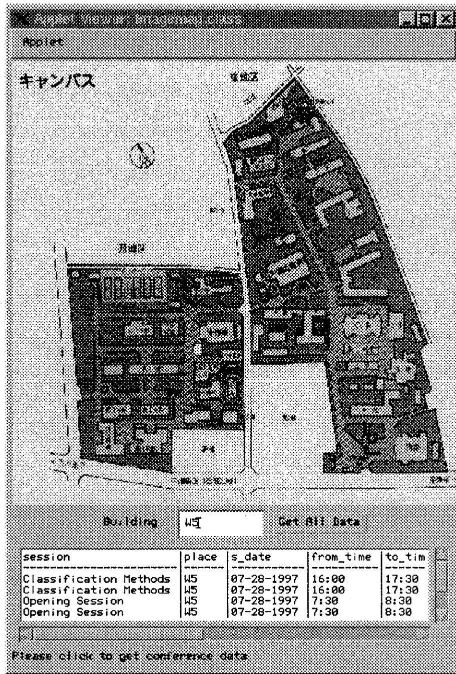


Fig. 7 A java applet client application.

applet that acts as a Web-embedded client application accessing the conference information from a relational database server. Users can load this application from its WWW server and access conference information held in each building by clicking at the building area in the map. Then, the Java applet sends the building code to access the conference information and shows it in the text area.

The interface definition of this Web-embedded client application is shown in **Fig. 8**. This interface definition defines a new class named SessionTitle. It is managed in database confDB-Web at the MIR whose e-mail address is ACC1@MIR1.is.uec.ac.jp. This interface contains a *body* clause that specifies a fixed number of arbitrary attributes. In this example, it has 7 attributes corresponding to the output displayed in the text area: **session, place, s_date, from_time, to_time, title and presenter**.

In contrast to the class definition of object-oriented programming that declared data type for each attribute, our interface definition declares a *domain* of data type for each attribute. In this example, a domain of session, title and presenter are string. The domain of place is buildingCode. The domain of s_date is Date and the domain of from_time and to_time are

```
database confDBWeb
address ACC1@MIR1.is.uec.ac.jp
class SessionTitle inherit wdbST
body
 session        string,
 place          buildingCode,
 s_date         Date,
 from_time      Time,
 to_time        Time,
 title          string,
 presenter      string,
method
public:
static string getByPlace(buildingCode $place);
boolean morningSession();
private:
OID  new(string $session,buildingCode $place,
         Date $date,Time $from_time,
         Time $to_time);
implement
sub new {
    my $package = shift;
    my $this;
    $this->{session} = shift;
    $this->{place} = shift;
    $this->{s_date} = shift;
    $this->{from_time} = shift;
    $this->{to_time} = shift;
    $this->{title} = shift;
    $this->{presenter} = shift;
    bless $this;
    return $this;
}
sub getByPlace{
   my $package = shift;
   my $place   = shift;
   return ``Place=$place'';
}
sub morningSession{
 ...
}
interface
Rule:
buildingCode($X)
 :- checkCode($X,$Y),output($Y).
checkCode($X,$Y)
 # If format of $X == XXXYYY set $Y = $X
 # XXXX is string and YYY is number,
 :- pattern($X,(\w+)(\d+)),set($Y,$X).
checkCode($X,$Y)
 # if ($X == XXX-YYY) set $Y = &chCode($X)
 :- pattern($X,(\w+)-(\d+)),
    set($Y,'&chCode('.$X.')').
External:
  chCode(input),chCode.pl,W-5,W5
Default: W5
Comment:
 Please Enter Building Code as follows:
     WX(X is number from 1 to 9)
Extract:
 # function name,program name,input,output
 bldName(input),bldName.pl,W5,W
 bldNo(input),bldNo.pl,W5,5
...
endclass
```

Fig. 8 An interface definition of the Java applet client application.

Time. Each domain has a *rule* that can be invoked for validating input data to solve the schematic mismatch problems. We will describe it in detail later.

In a *method* part of the interface definition, we have to provide the signature of all operations which represent associated methods available by this client. The implementation of each method is coded in an *implementation* part in Perl language syntax. The associated methods

of the method part are divided into 2 types; a *public* and *private* types. Users can directly invoke only the methods of the public type. In this example, getByPlace() method requires one argument and its data must agree with buildingCode rule.

The *new()* method is a constructor method. It is used to instantiate a Persistent Perl object of an interface definition.

The *static* keyword in the method signature means that this method will be used as a class method for generating the native command of the Web-embedded client application.

In case of using the global standard data format in cell design, the conversion between the standard format used in the cell and the native format used in client applications is processed in a wrapper component. The wrapper performs this conversion when invoking the *new* constructor or the native command.

The method that is not static method and returns boolean is a filtering method manipulating the object of Persistent Perl. We call this type of method a *boolean method*.

Let us now complete an example object in our common data model. Turn our attention to object presentation in the Persistent Perl. Suppose the following information that is exported from the above client application:

[Open Session, W5, 07-28-1997, 7:30, 8:00, Keynote Address, Gorge Miller].

An object in the Persistent Perl can be instantiated by passing this information to the constructor method as follows:
new SessionTitle ('Open Session', 'W5', '07-28-1997', '7:30', '8:00', 'Keynote Address', 'Gorge Miller').

We can represent the above value in an object as follows:

```
SessionTitle_1->session = 'Open Session'
SessionTitle_1->place = 'W5'
SessionTitle_1->s_date = '07-28-1997'
SessionTitle_1->from_time = '7:30'
SessionTitle_1->to_time = '8:00'
SessionTitle_1->title = 'Keynote Address'
SessionTitle_1->presenter = 'Gorge Miller'
```

SessionTitle_1 is Perl's associated variable reference. From the point of view of object, it acts as object identifier (OID). The Perl object that we use as object-relational model also supports many object-oriented abilities such as nested collection and so on. The detail can be found in Perl reference manual[3].

Now, we will describe to how methods are invoked in the Persistent Perl. As shown above, we have two types of method that have different syntax forms for invoking. The syntax form of a *static* method invocation is *ClassName→METHOD (ARGUMENTS)* while the syntax form of an *instance* method invocation is *OID→METHOD (ARGUMENTS)*. Here is an example of two uses of such a method:

```
SessionTitle->getByPlace('W5')
SessionTitle_1->morningSession()
```

In this example, the getByPlace('W5') is used to access information from the Java applet by invoking the *getConferenceData* ("W5") that is a method in the DA part of the Java applet shown in Fig. 7, while the morningSession() method is defined as a filtering method for evaluating whether the value of from_time attribute of the SessionTitle_1 object is less than 12:00 AM.

In an interface part of the interface definition, code of a rule for each domain is implemented in this part. Each rule can be divided into the following parts:

Rule: This part declares the rule for a domain used by the element such as an attribute or argument of methods in the interface definition. A rule is described as a series of *Horn clauses* and its flow control is the same as that of Prolog. It also provides predefined predicates performing matching capability of Perl. In this example, This rule is for the *buildingCode* domain. It supports two patterns of data presentation; W-X and WX where W is a string and X is a number. The WX is the default pattern for this client while the W-X pattern can be converted to this default pattern by using chCode() external function. In this case, there are only these two patterns used for representing the code of building. Therefore, for this example, we use **direct** conversion, which is the second method of cell design described in Section 2.2.

External: This part declares a group of functions that are provided by the information resource for converting the other data-pattern supported by this rule to the default pattern. In this example, chCode() function is defined in chCode.pl file. It changes data pattern from W-X to WX. The last two items are shown the example of input argument data (W-5) and returned data (W5).

Comment: This part declares the message shown to user when he is going to enter data for the argument of this interface rule. It is useful

for explaining some information for user during query-command generating process.

Default: This part declares a sample of data of its domain. The mediator passes this data to the other interface rule for execution to figure the schematic conflict out and inserts a conversion function to solve that schematic conflict for making the integration.

Extract: This part declares a group of functions that extract the sub-component of the default pattern. In this example, this part provides two functions: bldName and bldNo. They are used to extract a building name and a building number from a building code respectively. The method description is the same as that of description shown in the **external** part.

It is important to note that the interface definition shows that the output information and access methods of the client application are wrapped into the body part and the method part of the interface definition respectively. In case that developers want to enhance some capabilities of object filtering, they can code the boolean method part of the interface definition. Moreover, the data validation for solving the schematic conflict problem can also be described in the interface part.

3.2 Mediator on Mobile Side

In this subsection, we describe the OO-SQL syntax, the structure of the script program and a OO-SQL generator of the mediator component on a user's mobile unit.

3.2.1 OO-SQL Query Language

To help mobile users describing their task of integration using declarative expression, we modify the SQL query language to create our query language. We call our query language *OO-SQL*. The OO-SQL is not a new query language. Indeed, OO-SQL can be seen a query language that is targeted to our object-relational model. OO-SQL is similar to SQL[10] query language but we just change the data representation to correspond with Perl's object and add support for method invocations in a condition clause. A query in OO-SQL has the following syntactical structure:

select *target-attributes*
from [*object-variable*] in [*class-name*]
source [*class-name*] of [*database-name*]
    on [*address*]
where [*object-condition* or *method-call*]

Target-attributes is a list of required attributes. **From** clause contains a list of *object-variables* of classes. *Object-variable* is bound to object collection in *class-name*. **Source** clause contains the address and database of the class specified in **from** clause. **Where** clause contains query conditions. The method call syntax used in **where clause** is shown in method invocation of the Section 3.1.

Header	TOC	Object and Service	Query Command

Fig. 9 The structure of filtering script.

The OO-SQL as well as knowledge written in Persistent Perl is converted into a script program that has a structure shown in **Fig. 9** as follows:

The script header: The part is used to initialize the environment for starting execution.

The table of contents (TOC): It consists of indexes of resources required to accomplish a specific task. With the TOC, the wrapper can provide the abilities to collect/apply the services between each information source/MIR when it migrates from place to place.

The data objects and additional functions: The data object and additional functions that are accumulated from each information source/MIR will be saved into this part.

The query command: This part is query command that was generated by the user.

3.2.2 Mediator

The goal of the mediator on a mobile computer is to help a mobile user generate an OO-SQL command. The mediator explores information included in interface definitions, interacts with the user and provides this information for generating the OO-SQL command in disconnected state. To explain the behavior of the mediator running behind its user interface, we use the following example.

Consider a situation that our university holds a conference and wants to provide participants with information about the university. Assume that laboratories in our university are opened for visiting during the conference being held and a user wants to visit the laboratories in the same building where a session is held when he has free time.

This example requires two client applications. The first one is the java applet that its interface definition of session information is described in detail at the previous subsection. The sec-

ond is a Web-embedded client application that accesses laboratory information in a RDBMS through a CGI program. Let us show the interface definition of this laboratory information briefly as follows:

```
...
class LabBuilding
body:
  building     BdDomain,
  floor        int,
  lab          string
method
public:
  static string getByBuilding(BdDomain
                                     $building);
...
private:
...
inplement
...
interface
Rule:
BdDomain($X)
 :- checkCode($X,$Y),output($Y).
checkCode($X,$Y)
 # if XXX-YYY set $Y = $X
 :- pattern($X,(\w+)-(\d+)),set($Y,$X).
checkCode($X,$Y)
 # if XXXYYY set $Y = &changeCode($X)
 :- pattern($X,(\w+)(\d+)),
    set($Y,'&changeCode('.$X.')').
External:
 changeCode(input),changeCode.pl,W5,W-5
Default: W-5
Comment:
 Please Enter Building Code as follows:
   WX or W-X(X is number from 1 to 9)
Extract:
 # function name,program name,input,output
 buildingName(input),buildName.pl,W-5,W
 buildingNo(input),buildNo.pl,W-5,5
```

The rule part of this interface definition handles the building code in the W-X format. This rule can convert the building code from the WX to the W-X format using a *changeCode* subroutine. This rule uses the direct conversion for resolving the schematic conflict problem. Now, consider an OO-SQL command that selects morning sessions held at a W5 building and the laboratories that are in the same building of the sessions. It can be written as follows:

```
select S->{session},S->{place},S->{s_date},
       L->{building},L->{floor},L->{lab},
from S in SessionTitle, L in LabBuilding,
source SessionTitle of confDBWeb
       on Acc1@MIR1.is.uec.ac.jp,
       LabBuilding of uecdbWeb
       on Acc1@MIR1.is.uec.ac.jp,
where SessionTitle->getByPlace('W5')
and  S->morningSession()
and  LabBuilding->getByBuilding(
          &changeCode(S->{place}))
```

We assume that the information sources used in this example are managed in the same MIR. Also, the interface rule is the direct conversion rule because there are only two formats

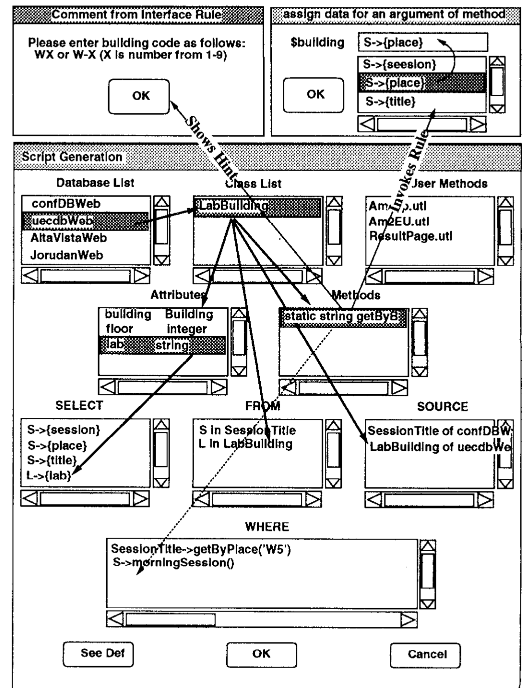


Fig. 10 User Interface of Mediator.

in this MIR. Then, in this query, the user gets the morning session data in the building 'W5' by using the first and second condition. The laboratory information in the same building of the sessions is accessed by the third condition. The &changeCode function is inserted to convert data format from WX to W-X by BdDomain interface rule.

Recalling to the method invocation of Perl object, the execution in this query command can be described as the following steps: Firstly, the static command SessionTitle→getByPlace('W5') is converted into the native command of the Web-embedded client application corresponding to the *SessionTitle* class. In this case, it is the java applet we showed earlier. Secondly, objects in *SessionTile* class that created by the first condition will be iterated by S *object variable*, OID→morningSession(), to filter the objects that satisfy the morningSession boolean-function. Thirdly, the filtered objects will be used as condition to join with laboratory information. Finally, a project operation will be performed to get only the *target attributes*.

In order to build the above query, a user is helped by the mediator on his mobile unit. **Figure 10** shows the user interface for this query generation. According to Fig. 10, the user inter-

acts with the mediator through a user interface by the following step: (1) First, when the user chooses the database name, he can list interface definitions registered earlier. (2) Next, he selects an interface definition in the list, the mediator will show the body and public method parts of the interface definition into attribute and method windows respectively. Items in **from** and **source** clauses are also created. (3) Next, attributes selected by the user will be created as target attributes in **select** clause. (4) Next, in case that the user chooses a method from method window, the mediator will check the number of arguments as well as their domains and ask the user a input data for each argument. (5) For each domain of the argument, the mediator invokes the interface rule that is associated with the current argument, hints the comment part to the user and waiting for the user's input. (6) After the data is inputted, it will be passed to the interface rule for schematic conflict checking. In case that the user inputs an attribute-name that it is not the real data value, the mediator reads the domain of that attribute-name and passes its default value of the interface rule of that attribute to solve the schematic conflict. If the interface rule can not solve the conflict, the extract functions will be provided to the user for making decision to integration. In this way, the user can generate OO-SQL script by the help of his mediator.

From the above query command, when the user selects the *getByBuilding* method of the *Laboratory* interface definition that has one BdDomain argument, the mediator opens two windows for showing the comment of BdDomain and waiting for the input for $building argument as shown in the above Fig. 10, respecitvely. Suppose that the user inputs S→{place} as input data into the field. Because the domain of S→{place} is *buildingCode* and its default data is *W5* (shown in body and interface rule part of the interface rule in subsection 3.1 respectively), the mediator then invokes the BdDomain interface rule (shown in this subsection) and simulate the execution by passing default value W5 to the BdDomain interface rule. As a result of interface rule execution, the mediator inserts the *changeCode* function that is used for converting pattern of building code and generates the method invocation as the condition of OO-SQL command.

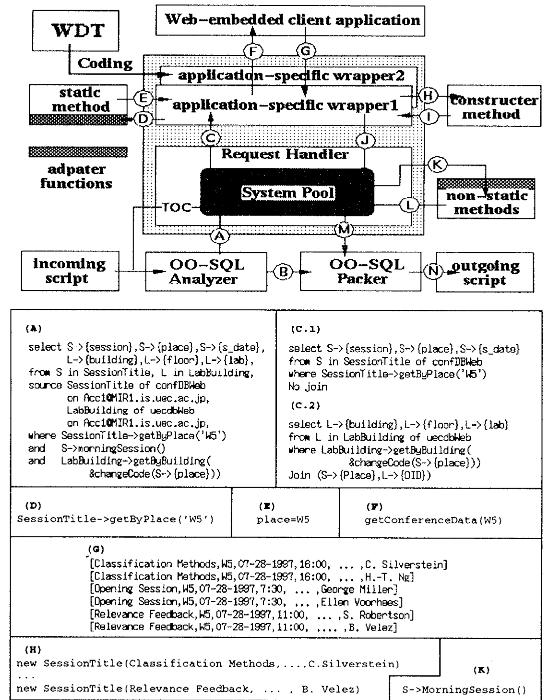


Fig. 11 A wrapper architecture.

4. Wrapper Architecture and Query Execution

In this section, we describe how a wrapper executes a query command of an incoming script program. To understand the query execution more clearly, we discuss the example with the query command shown earlier. When the query command together with the necessary information that are formed as a script program arrives at the MIR, the wrapper will perform as shown in **Fig. 11**:

OO-SQL Analyzer: The OO-SQL Analyzer checks the incoming command and divides it into two parts: a related command (A) and an unrelated command (B). The related command is the sent for execution while the unrelated command is sent to OO-SQL Packer. In this case, all information is in the same MIR. Therefore, no unrelated command is produced.

Request Handler: The request handler is one of the important component of wrapper. It controls the execution process. After checking the **TOC** (Table of Contents) of an incoming script, it loads the necessary services from its library, the additional services and objects packed from the incoming script into the system pool. In this case, it will load *changeCode*

function from system library of this MIR. After initializing the environment, it divides the command into static and boolean methods and passes the execution processing with static command method (C) to the application-specific wrapper. This command is concerned with two Web-embedded client applications; C.1 and C.2. The application C.1 is first executed and the boolean method (K) will be applied to filter the result in later.

The static method is executed (D) and its returned result (E) is then mapped and converted to the native command (F). After execution is performed, the output stream (G) is returned. The required string data is extracted from the output stream and passed to the constructor method (H) defined in the interface definition to create objects (I) in common object model. After creating the objects, the application-specific wrapper invokes join-operation, registers the result of join into system pool (J) and passes the execution to the request handler for invoking filter methods (boolean method (K)) to filter the qualified objects (L). In this case, the first object and second object are filtered out because their *from_time* values equal 16:00.

After finishing this execution, the request handler will start the next execution (C.2). The process execution flow is the same as the previous one that we have just described.

To accomplish the above task, we provide a WDT (Wrapper Development Tool) for developing a wrapper. The developer can write the application-specific wrapper with little effort to control the execution of client-application call, data model conversion and join process that we have described above.

OO-SQL Packer: The OO-SQL Packer packs the result objects and the necessary services (M) from the system into TOC of script. It updates OO-SQL with the unrelated command (B) into the script program whose format is shown in Fig. 9 and sends this script, if necessary, to the next information source/MIR.

The processing done in the above modules will continue until the filtering script completes the execution. The result of the execution will be sent back to a server specified by the user.

5. Preliminary Experiment

To test the realization of our approach, we have implemented and tested the wrappers for the following information sources shown in **Table 1**. To test the heterogeneous Web-embedded client applications, we have used all three types of the clients shown in Fig. 3. The first source is the a collection of HTML files which is accessible through a Persistent Perl mobile script communicating with the server written by Persistent Perl. The second is the java applet client used in Fig. 7. The others are Web-embedded client using the CGI interface. The Altavista server is running at http://www.altavista.com/ and the server of transportation information is running at http://www.jorudan.co.jp/.

Table 1 Information sources and their Web-embedded client apllications.

Information Source	Client	Backend	MIR
HTML Files	mobile script	Perl Program	MIR1
Conference DB	Applet	Postgres	MIR1
Laboratory DB	CGI	Postgres	MIR1
Homepage in UEC	CGI	Altavista	MIR1
Event DB	CGI	Postgres	MIR2
Transportation	CGI	unknown	MIR2

MIR1 and MIR2 are running on the UNIX workstations. The user carrries a mobile computer, which has a user interface implemented in Java and running on the Hotjava browser. The user interface lets a user generate a filtering script (Fig. 10), sends it to MIRs, browses the result and so on.

Recall the query we used as an example in Section 3.2.2. **Figure 12** (left) shows the Web-embedded client application of the second information source of Table 1. Now we also add the fourth information source of the Table 1 and pass the laboratory name as a keyword to find out the URLs of homepages in our university. With the user interface of the mediator shown in Fig. 10, we can generate an OO-SQL query command as follows:

```
select S->{session},S->{place},S->{s_date},
       L->{building},L->{floor},L->{lab},
       Se->{urlList}
from S in SessionTitle, L in LabBuilding,
     Se in SearchEngine
source SessionTitle of confDBWeb
  on Acc1@MIR1.is.uec.ac.jp,
       LabBuilding of uecdbWeb
  on Acc1@MIR1.is.uec.ac.jp,
       SearchEngine of altaVistaWeb
  on Acc1@MIR1.hol.is.uec.ac.jp
where S->confirm(S->{session},S->{place},
      S->{s_date},S->{title})
and   LabBuilding->getByBuilding(
          &changeCode(S->{place}))
and   SearchEngine->searchByKey(L->{lab})
and   &resultPage('Building',L->{building},
      'Floor',L->{floor},'Lab',L->{lab},
      'Session',S->{session},
```

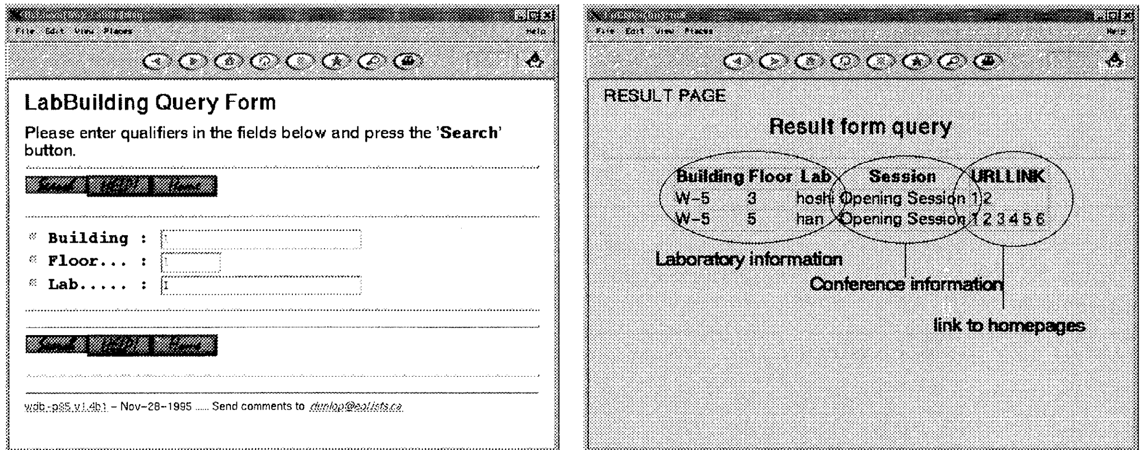


Fig. 12 LabBuilding Web page and execution result.

```
'URLLINK',Se->{urlList})
```

In this example, we also show a *confirm* function that is one important feature for information integration. With this function, the wrapper stops the execution as the check point and creates the intermediate result. The user can select the subset of result as the condition of the next task and send it to the network to restart the execute again. This feature is appropriate for not only generating interaction to receive decision from the users but also preventing the explosion of data from the join process. Finally, the last condition is the user-provided resultPage() method that converts data in Persistent Perl object into HTML file. The final result of the execution is shown in Fig. 12 (right).

We have also tried other experiments in various combinations of the Web-clients shown in Table 1. Based on our observations, the interface definition that contains services description and executable rules is suitable for describing heterogeneous Web-clients as shown in Fig. 3 and for providing the information to generate customized application on the mobile computer in the disconnected state. About the schematic conflict problem between different cells, the extract functions can solve the problem at some level. However, using the well-known domain is a preferable solution for the schematic conflict problem. Comparing with the other software component architectures such as Active-X or Java Bean, our work provides the architecture and tools for mobile users to be able to easily create customized applications for integrating information from heterogeneous Web-embedded client applications in the remote programming style.

6. Summary and Discussion

In this paper, we have described the architecture for mobile users to integrate information from heterogeneous Web-embedded client applications in the remote programming style. By wrapping heterogeneous Web-embedded client applications in interface definitions, the mobile users can see these applications as components that advertise their services. With the mediator on mobile side, the information in these components is automatically explored and used to create "customized applications" by the mobile users in disconnected state. This is our originality different from various works of generic software components that are integrated to create an general applications working in distributed object-oriented frameworks.

Our approach does not require mobile users to describe the details of integration in a query command. Instead, our mediator on a mobile unit tries to help the users in providing hint information, solving the schematic conflict problem and generating a query command.

With this approach, the mobile users load the new interface definitions and integrate them with the existing ones to create new customized applications while they are moving around the network. Moreover, the customized applications created to suit the specific requirement of an application in any area can be exchanged together in a group of mobile users, thereby enabling more widespread network-services utilization. With the cell design and the mediator on mobile unit, our approch provides mo-

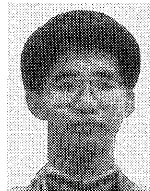
bile users with a capablity for generating integration view when comparing to the TSIMMIS and HERMES, and with a specialized and easier integration environment when comparing to CORBA and DCOM.

In the future, we will investigate the possibility of introducing a template-based specification for developing a wrapper. We also investigate the possibility of wrapping the GUI part to a much wider range of Web-embedded client applications.

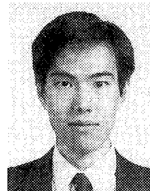
References

1) Armstrong, T.: *Designing and Using ActiveX Controls*, p.630, M&T Books, New York (1997).
2) Brookshier, D.: *Java Beans Developer's Reference*, p.733, New Riders Publishing (1997).
3) Wall, L.: *Programming Perl*, p.646, O'Reilly and Associates (1996).
4) Chawathe, S. et al: The TSIMMIS project: Integration of heterogeneous information sources, *Proc. 100th IPSJ Tech. Rep. DBS* Tokyo, Japan pp.7–18 (1994).
5) Wisut, S.T.: An Information-Retrieval Architecture for Mobile Computers based on a Persistent Script Language, Master's Thesis, p.61, Dept. Information System, The University of Electro-Communications (1995).
6) Wisut, S.T., Ohmori, T. and Hoshi, M.: An Information-Retrieval Architecture for Mobile Computers based on a Persistent Script Language, *IPSJ Tech. Rep. DBS*, Vol.96, No.109–45, pp.269–274 (1996).
7) Subrahmanian, V.S. et al.: HERMES: A Heterogeneous Reasoning and Mediator System, available at http://www.cs.umd.edu//projects/hermes/publications/authors/all.html
8) Lange, D.B.: Java Aglet Application Programming Interface (J-AAPI) White Paper, available at http://aglets.trl.ibm.co.jp/JAAPI-whitepaper.html
9) 山崎重一朗：エージェント指向のスクリプト言語の最新動向「Telescriptを中心として」, *Proc. Advanced Database System Symposium '95*, Tokyo, Japan, pp.59–67 (1995).
10) 西尾章治郎：実践 *SQL*, p.302, アスキー出版局 (1996).
11) Vinoski, S.: CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments, p.12, available at http://www.acl.lanl.gov/CORBA/#DOCS

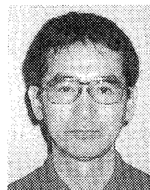
(Received September 2, 1997)
(Accepted February 2, 1998)



Wisut Sae-Tung is a docteral student in graduate school of information systems at the university of Electro-Communications (UEC). He received a BE in electrical engineering from Chulalongkorn University, Thailand, in 1988 and ME in information engineering from UEC in 1995. His research interests include heterogeneous information system, mobile computing, programming language and database technology.



Tadashi Ohmori received Dr.Eng. degree in 1990 from The University of Tokyo. Since 1994, he has been an Assistant Professor for Graduate School of Information Systems, The University of Electro-Communications. His research interests are data engineering as well as database-oriented middlewares for advanced applications.



Mamoru Hoshi received the B.E., M.E., and Dr.E. degrees in mathematical engineering from the University of Tokyo, in 1967, 1970, and 1985 respectively. During 1970–1978 he was with the Electrotechnical Laboratory, Tokyo, Japan. During 1978–1992, he was on the Faculty of Engineering of Chiba University. He is now a Professor of the Graduate School of Information Systems, The University of Electro-Communications, Tokyo, Japan. His research interests include algorithms and data structures for searching, image processing, and computer graphics.