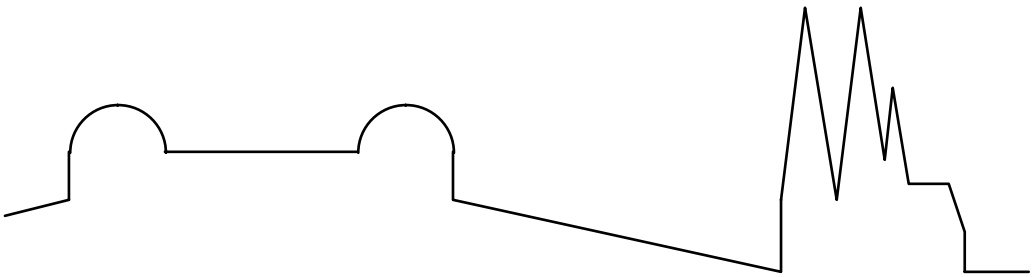
Accessing web forms from an object-relational database system

Johan Petrini



Thesis for the Degree of Master of Science
Majoring in Computer Science, 10 credit points
Autumn 2001

Department of Information Science
Computer Science Division
Uppsala University
P.O. Box 513
S-751 20 UPPSALA
Sweden

Abstract

A main memory object-relational database system, AMOS II, has been developed at Uppsala Database Laboratory (UDBL). The system provides common database facilities and a powerful query language but also, through it's mediator-wrapper approach, features for the combination of data from heterogeneous data sources.

The AMOS II query processor is extensible through a generalized foreign function mechanism. Currently AMOS II has interfaces to the programming languages C, Lisp and Java implying that applications working against AMOS II can be developed in one of these programming languages.

Unfortunately, accessing information from the web cannot be done in the same way as using conventional databases. For the extraction of data from the web there exists Internet wrapper toolkits to specify programmatic interfaces to more or less well structured web sources.

The thesis is about developing an extension to the existing AMOS II system, ORWIF (Object Relational Wrapper of Internet Forms), that facilitates the combined access to data from the web with data from other data sources e.g databases. In the ORWIF project data is extracted from a web form using foreign functions, existing Java libraries and a publicly available Internet wrapper toolkit. A description and comparison among three existing Internet wrapper toolkits is also included.

My solution offers a flexible and easy way for users to access and analyse information retrieved from web forms and combining it with data from other sources through an OO mediator database system. For example, users can do their own "price running" with ORWIF.

In my work I addressed the issue of optional input fields in web forms and how this can complicate extracting information from web sources. The problem was solved through delegating the responsibility of handling these, so called, omitted parameter values to the code of the foreign function.

Table of contents

1. Introduction

With the origin of high speed communication networks, such as the Internet, computing environments have become increasingly distributed. It is possible to utilize information from several heterogeneous sources of information located at different places. However, when trying to combine data from such data sources technical difficulties emerges. These difficulties can be handled through the use of a mediator [5].

An example of such a mediator system is the AMOS II system developed at Uppsala Database Laboratory (UDBL). AMOS II is a main memory object-relational database system that integrates multiple potentially different and distributed data sources [5] using the *wrapper-mediator* approach [4]. With a query language called AMOSQL, users can execute object-oriented queries over these heterogeneous data sources [4].

The query processor in AMOS II is extensible through a generalized foreign function mechanism. Currently AMOS II has interfaces to C, Lisp and Java meaning that applications working against AMOS II can be developed in several programming languages.

Information in heterogeneous data sources today is typically split between structured data in traditional databases and the massive amount of unstructured information available over the web. The benefit of achieving interoperability among these different information sources is compounding [20]. Unfortunately it is not feasible to access information from the web in the same way as using conventional databases. For example there are no standardized query interface to web sources. Also the content of web sources, in the form of web pages, is often not well structured. Another problem is that web sources is highly dynamic and can change its content and structure momentarily.

For the extraction of data from web pages there exists special purpose software, so called Internet wrapper toolkits, to specify programmatic interfaces to more or less well structured web sources [11].

As concluded from the above it is not a trivial task to integrate and utilize information from web sources and traditional databases in an effective way. The purpose of my project is to develop an application that can process queries in AMOSQL combining data from different web forms with data from regular databases or other data sources. The integration mechanism should be implemented using foreign functions, existing Java libraries and an existing publicly available Internet wrapper toolkit. Today there are several Internet wrapper toolkits available for anyone to use. A description and comparison of three of them is to be presented.

The outcome of the project shows that AMOSQL queries can be specified combining data from an AMOS II database with data retrieved from the chosen web form.

The method used in this project has been to implement a prototype, Object Relational Wrapper of Internet Forms (ORWIF), fulfilling the aim set up in the problem specification. Several technical problems arouse during this process. After carefully discussing and solving these problems the solutions were applied to the prototype. One of the main problems was how to map the nature of a specific web form to an AMOS II foreign function. The practical work was complemented with a study of existing literature, research and solutions concerning Internet wrapper toolkits.

The structure of my report can best be understood divided in two sections were the first part provide the reader with the underlying theory on which I have based my work. Different database management systems (DBMS) and their evolution is presented along with an introduction to the wrapper-mediator approach concerning gathering and integrating information from heterogonous data sources. Special attention is given to the difficulties in wrapping web sources and the software developed to aid the programmer in this matter e.g Internet wrapper toolkits. This is followed by an overview of the AMOS II system including a short summary of the AMOS II Java interface. The second part, containing a presentation of my work, explains the architecture of ORWIF, provides the reader with some implementation specific details, and shows some examples of how ORWIF works. A description and comparison among a few different Internet wrapper toolkits is also presented. The report is completed with a discussion and subsequent conclusions.

2. Background

This chapter gives an overview of the evolution of different database systems, the wrapper-mediator approach to integrate information from heterogeneous data sources and technologies used for accessing information from the web such as Internet wrapper toolkits.

2.1 Object-relational databases

Traditional databases such as relational, network, and hierarchical have been successful in offering database technology required for many traditional business database applications. The relational database management system (RDBMS) is the most frequently used DBMS in the commercial field due to its simplicity, query language, and well understood underlying mathematical theories [15].

New more complex database applications for example databases for computer-aided design (CAD) and manufacturing (CAM), bioinformatic systems for storing gnome information, geographic information systems, and multimedia databases have characteristics that differ from those of traditional applications. This leads to new requirements on databases such as the possibility to model more complex objects, support for longer transactions, new data types for storing images or large textual items(extensibility) and the need to define non-standard application specific operations on data types [23].

The first object database management system (ODBMS) prototypes developed in the early 1990's were proposed to solve some of these problems. They enabled the designer to specify both the structure of complex objects and the operations that can be applied over these objects [23].

In an object database the information is stored as objects. Each object is uniquely identifiable in the database by an object identifier and can be described as a real-world entity that contains information about its current state (attributes) together with the actions that can be taken on the object (methods or operations). OO databases provides a tight interface to OO-programming languages like C++, Smalltalk and Java extending them with DBMS primitives e.g functionality for persistent storage of data structures. Integrating DBMS logic in OO-programming languages also facilitates for object-oriented programmers to create OO-database applications. Popular ODBMS's today are Jasmine, Gemstone, O2 and Object Store [23].

However there are some drawbacks with ODBMS's when comparing them to RDBMS's. For example, unlike relational databases which follows some well-defined and accepted guidelines, the standard query language (SQL), there have been an absence of consensus concerning data models and a query language for the ODBMS. This has resulted in poor portability and interoperability among applications accessing the ODBMS but also in primitive query facilities. In recent years the object database management group (ODMG), a consortium of ODBMS vendors, has specified standards for an object model, an object query language (OQL), and the bindings to object-oriented programming languages. The ODMG's latest object database standard, version 3.0, was published in January 2000 [23].

Stonebraker gives a classification of a new class of database management system – the object-relational DBMS (ORDBMS). He views the object-relational DBMS as a class of database management systems that attempts to combine the relational DBMS and the object DBMS. To clarify what he means Stonebraker describes the different requirements to data storing based on querying facilities and data modelling facilities by introducing the application matrix below:

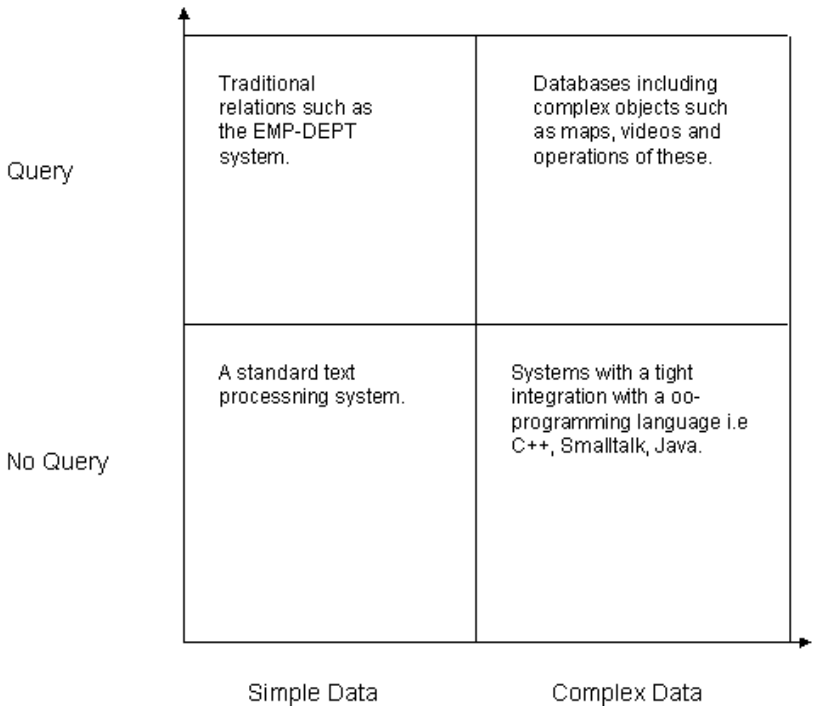


Figure 1: Stonebrakers application matrix [15].

Stonebraker argues that the existing DBMS cannot cope with complex data at the same time as giving good querying facilities. A fully object-relational database should according to Stonebraker meet the requirements of the upper right of the matrix [15].

The idea behind object-relational databases is to extend the functionality of established relational databases, with their simplicity, query language, and well understood underlying mathematical

theories, to meet the demands from new database applications as mentioned earlier. These extensions can be summarized as[23]:

- Support for user defined abstract data types(ADT's) and operations over these. This feature allows the user to model and manipulate complex objects. The use of ADT's make the relational system behave like an ODBMS and drastically cuts down the programming effort needed compared with achieving the same functionality with SQL embedded in a programming language.

- Object-orientated features of the declarative standard query language (SQL). SQL-99, also known as SQL3 provides concepts such as ADT's and inheritance were inheritance is the ability to create new objects based on existing ones. SQL-99, is also considered as a beginning to a standardization of object-relational databases.

- Support for an application programming interface to data blades. Data blades (data cartridges, data extenders) are database 'plug ins' that provide predefined application specific data types and operations over these. Examples of data types can be two-dimensional data types, image data types, text data types etc.

The first commercial object relational databases were presented in the late 1990s. Before then many prototypes had been developed e.g Iris (HP), Postgres (Berkeley), Starburst (IBM). Today Informix is the market leader in object relational databases. Other products in the ORDBMS market are Oracle 8.x, Universal Database (DB/2 Extenders) and UniSQL/X. An important distinction is to be made between products working as the RDBMS with enhanced object-oriented functionality, and others like UniSQL/X developed from the beginning as an ORDBMS product [23].

In an ORDBMS information is stored in tables of rows and columns just like in a ordinary RDBMS. Tabular entries can, with the support for ADT's, contain richer data structures. Operations and functions associated with new data types can be used to index, store and retrieve records based on the content of the data type. Since the information is stored in tabular form the ORDBMS must also have some functionality for translation between objects and tables [24].

2.2 Mediators and wrappers

The development of world wide high speed communication networks has led to a more distributed computing environment with increased access to data from different nodes in the network. To utilize such information there is a need for new technologies that enable the integration of data from several, and sometimes different, data sources and present them in a way that makes sense for the user. However, when trying to combine data from distributed heterogeneous data sources several problem arise due to the fact that the data sources can have different representations of same 'real world' data. Data sources may use different data models and languages and might contain equivalent, conflicting or complementary data which must be reconciled before they are displayed to the user [5].

Wiederholt [5] describes the situation in terms of interfaces between user workstations and database servers. He means that traditional passive interfaces which only defines communication protocols and formats in term of database elements does not qualify to deal with representational

problems of existing data sources. Instead he proposes the use of active interfaces implemented by software modules called *mediators*. The mediator is thought of as an intelligent middle layer between the application and the database making the application independent of the data sources, helping them to communicate in a more meaningful and efficient way. The functionality needed to accomplish this is for example transformation and sub-setting of databases using view definitions, methods to access and merge data from multiple databases, support for abstraction and generalisation of underlying data, and methods to deal with uncertainty and missing data because of incomplete or mismatched data sources [5]. Wiederholt defines a mediator as:

"software modules that exploits encoded knowledge about some sets or subsets of data to create information for a higher layer of applications".

He also identifies three layer in a conceptual model of the mediator architecture:

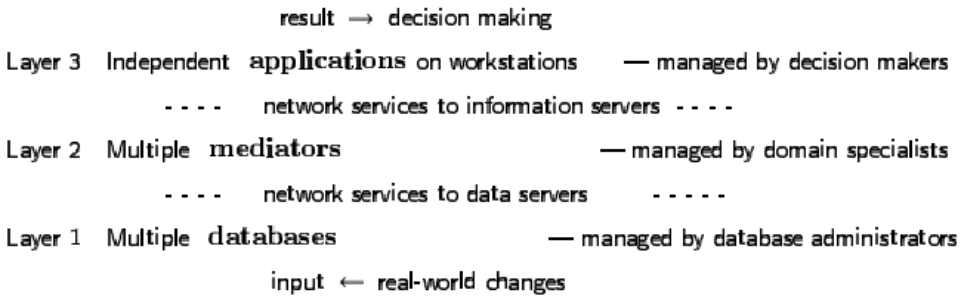


Figure 2: Mediator architecture [5].

Wiederholt introduces the wrapper-mediator approach building on the concept of mediators. It divides the functionality of a data integration system into two parts. The *wrapper* part offers access to data from different data sources. It handles the responsibility of understanding the internal logic of each data source. Generally, a wrapper can be looked upon as a procedure designed for extracting information from a particular information source and delivering the content of interest in a self-describing representation. The mediator part has mechanisms for the processing and combination of accessed data. This makes it possible for the mediator to provide a coherent view of data in the wrapped data sources so that other applications can easily access that data [4].

Wrappers are mainly used in the database community and the web environment. They are useful means of enabling expressive queries over data that was not necessarily designed for querying and wrappers also facilitate the integration of data from multiple possible heterogeneous sources. Examples of functionality for wrappers could be accessing relational databases through Open Database Connectivity (ODBC) or accessing data sources in the form of web pages through the Hypertext Transfer Protocol (HTTP)[4]. Also distributed systems built in accordance to the Common Object Request Broker Architecture (CORBA) can be wrapped [13]. A wrapper concept has recently been introduced as a new part of SQL, called SQL/Management of External Data [6]. SQL/MED contains language constructs for the retrieval of data from a foreign server using a foreign-data wrapper [6].

2.3 Internet wrapper toolkits

Internet can be seen as a gigantic source of information in the form of web pages. Today 80% of the information published on the Internet is generated by underlying databases using web forms as query language and the Hypertext Markup Language (HTML) as a tool for displaying the result [7]. To make integration of web sources, using the mediator approach, feasible wrappers are needed for web sources to be accessed. Such wrappers would accept a query from the mediator, fetch the relevant pages from that source, extract the requested information from the retrieved pages and return the result to the mediator. Essentially the wrapper should make the web source look like a database accessible through the mediator query language. The basic techniques used in database integration are simply applied to web source integration [9]. However, several problems arise when trying to access this kind of data compared to using conventional databases. To solve these problems so called Internet wrapper toolkits have been developed [10]. The ORWIF project described in chapter 3 uses the functionality of a wrapper toolkit to help extract information from web sources.

The difficulties when trying to access data from web sources can be summarized accordingly:

- There exists no standardized query interface such as ODBC or Java Database Connectivity (JDBC) for web sources. Web pages are primarily designed for human browsing rather than for use by an external application program. An information integrating system trying to access data from a web page have to work according to the same principle as a web browser. Queries are executed through submitting data from web forms.

- The result of a search engine query does not only have to contain the wanted data but also banners, pictures etc. To make the presentation of the information more clear such unnecessary information should be filtered away before displaying the result.

- The information on the Internet is often not well structured. Although the emergence of the eXtended Markup Language (XML) promises an increase of structured content there will for a long time be an enormous amount of less structured HTML pages out there.

In order to access data from web sources an information integration system needs some kind of specialized software to interface and process the external data so that it can be handled transparently in a uniformed and structured way. Such a software is often called Internet, or web, wrappers and can be developed manually by a programmer. This is however not so practical because of the following reasons [10]:

- Internet is highly dynamic. The content and presentation of web sources may change frequently and autonomously. Therefore flexibility is an important quality when trying to access web pages.

- The number of information sources of interest is normally very large.

- Newer sources of interest are added frequently on the web.

Instead Internet wrappers can be developed in various ways with the help of Internet wrapper toolkits. Internet wrapper toolkits specify programmatic interfaces to more or less well structured web sources handling both sending commands and extracting structured data from responses.

Further, it defines a web source wrapper by processing wrapper specifications, consisting of statements that connects to web sources and detects parts of the information to be extracted [11]. The extraction of data can be implemented using different approaches.

One uses the possibility to represent the HTML page as tag-based hierarchy according to the Document Object Model (DOM)[22]. DOM is an application-programming interface (API) for valid HTML and well-formed XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated. With DOM, programmers can build documents, navigate their structure, and add, modify, or delete elements and content. Anything found in an HTML or XML document can be accessed, changed, deleted, or added using the DOM with a few exceptions. The disadvantage with the technique is that building the hierarchy tree involves parsing the whole HTML page witch is an expensive and performance reducing procedure. The advantage is that the extraction mechanism often is quite robust meaning that extensive altering of the HTML source page can be done without affecting the functionality of the extraction mechanism.

Another approach utilises the power of pattern matching and ignores the HTML tag based hierarchy. Using pattern matching can in a more satisfying way handle inconsistencies in semi-structured HTML pages when it operates on a ‘character by character ’level instead of on a structural level.

An Internet wrapper toolkit can be a wrapper generator that generates code implementing a web source wrapper. It can also be a wrapper interpretater where the web source wrapper is specified as commands, which are interpretated at run time [11]. Both wrapper generators and wrapper interpretaters provide features such as fast prototyping and robustness to information-extraction on the web.

3. AMOS II

AMOS II is a wrapper-mediator object-relational DBMS. The AMOS II project was originally developed to demonstrate how applications and users could use an information system for combining and analysing data from different data sources. A data source could be a conventional database, text files and web pages. It could also be programs that collect measurements, or even programs that perform computations and other services. The data sources are distributed over a communication network such as the Internet. Engineering, telecom, and decision support are areas of application for this kind of architecture. Other systems with functionality related to AMOS II is DISCO[29], Garlic[30], and Multibase [31].

3.1 AMOS II architecture

The AMOS II approach to the wrapper-mediator software architecture is a distributed mediator system where several AMOS II mediator servers can communicate over the Internet via a TCP/IP based protocol. Other protocols are used for communication with non-AMOS II systems, e.g. data sources that communicate using ODBC or HTTP [4].

The core in the AMOS II mediator approach is the light-weight AMOS II mediator DBMS. The DBMS contains functionality for processing and executing queries over data stored locally but also external data sources. Applications and other AMOS II servers can access data from distributed, heterogeneous data sources through one or several AMOS II servers. So called OO mediation primitives are used to integrate and translate data in the external data sources to high level abstractions, OO views. This abstraction provide transparent access to and hides the details of the data sources from the user as well as the application programmer [4].

Wrappers are essential in order to handle external data sources. In AMOS II wrappers have previously been defined for ODBC, XML, and STEP/EXPRESS data sources. Even other AMOS II servers are treated as external data sources and are therefore also wrapped. A central thought in the development of the AMOS II mediator is that it can be customized for specific application areas. This means that the performance requirements on mediator databases can be very high for some application domains [4].

The architecture of AMOS II is illustrated below:

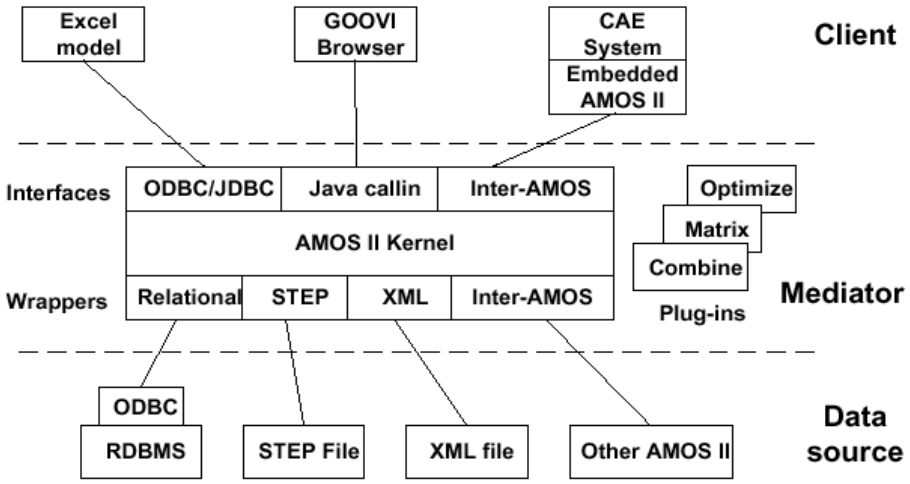


Figure 3: The AMOS II architecture [4].

The AMOS II DBMS is implemented as a main memory DBMS optimised to perform at it's highest in main-memory [2]. It can be used as a single-user database as well as a multi-user server to applications and to other AMOS II systems. AMOS II runs under Windows NT/2000 and Solaris and utilizes about 2 MB of memory [4].

3.2 AMOS II data model

All entities in the database are represented as objects and managed by the system. The two main types of object representations are literals and surrogates. Surrogate objects are characterized by having explicit object identifiers (OID's) and that they are created and deleted by the user of the system. Examples of the more complex surrogates could be objects representing real world entities such as a person or a car. A surrogate object can also be an meta object such as types or functions. The existence of meta objects permits the user to make queries involving the entire structure of a AMOS II database. Literals on the other hand are self-described, system maintained objects that does not have explicit OID's. Numbers and strings are examples of literals but they can also be collections of other objects such as vectors e.g 1-dimensional arrays or bags e.g unordered set with duplicates. Surrogate and literal objects persist in the database until they are no longer referenced by any other object or external system. The removal is handled by an automated garbage collector [4].

3.2.1 Data types

Objects in AMOS II are classified into instances of types. Types are organized in a supertype/subtype hierarchy with multiple inheritance. The set off all instances of a type is said to be the extent of that type and is a subset of the extent of it's supertype. An objects most specific type is the type specified in the creation of the object. A type set constitutes all types an object is an instance off and can, with the help of AMOSQL statements, change during different stages in the life of an object. This facility also enable the role of an object to change in the database witch is a very important aspect of database evolution. AMOS II also supports multiple inheritance meaning that one object can have more that one supertype at the same time. The AMOS II data model provides four categories of types: stored types, derived types, proxy types, and integration union types [25].

Stored types is the most common data type that the user of the system will come in contact and work with. The type definition is stored in the database and the instances of the types are managed by the user with the help of different AMOSQL statements. The general syntax to create a new type is [4]:

```
create type <typename>
```

It is also possible to make a new type inherit all properties and relationships of an already existing type. The general syntax to create a type as a subtype under a predefined super type is:

```
create type <typename> under <previously defined type>
```

Database evolution is supported in AMOS II through a mechanism to add or remove existing types from an object. Derived types, proxy types, and integration union types are used to provide the system with data integration features [4].

3.2.2 Functions

Functions in AMOS II models the semantics of objects e.g properties of objects, computations over objects, and relationships between objects [25]. The function itself consists of two parts, the

signature and the implementation. The signature defines the types and optional names of arguments and result parameters. The implementation describes how to compute the result given the argument values. Basic functions can be classified into different categories [4].

Stored functions represent properties, attributes, of objects in the database. For example, common properties of an object of type person are name and age. A call to the stored function name on such an object returns the current value of the attribute name. The general syntax for creating a stored function is [4]:

```
create function <functionname(type)> -> <return type> as stored
```

The function can now be used to set the value for one of the objects properties according to the general syntax [4]:

```
set <functionname><object instance> = <value>
```

Stored functions can also be used in a similar way to model relationships between objects.[4]

Derived functions are defined in terms of other predefined AMOSQL functions or queries. They cannot have any side effects, e.g they are not allowed to manipulate the database, and are compiled and optimised by AMOS II for later use [4].

Foreign functions are implemented in an external programming language [4].They are described more extensively in chapter 3.3.2.

Functions in AMOS II can be overloaded and have different implementations, resolvents, depending on their argument(s). This functionality implies the possibility to create a generic function applying to different object types. It's the task for the query compiler to apply the right implementation to a certain function call. The binding of a function name to the right resolvent of the function can be done at compile time called early binding or at run time called late binding. Early binding means that the system will try to use local variable declarations to choose the correct resolvents [3]. Amos II also supports late binding where the right resolvent at compile time still is unknown. Like types functions can also have extents. The extent of a functions is defined as the mapping between its arguments and its results [2].

Foreign functions *multidirectional* meaning that they can be executed when some results are known rather then just the arguments. This is possible when using binding patterns that indicate which argument or result in a given implementation that are known or unknown, respectively. Multidirectional foreign functions contribute to a higher degree of executable queries and also better query optimisation for the system.

A binding pattern is a string of b's and f's, indicating which arguments or results in a given implementation are known or unknown, respectively. Often the multi directional foreign function has one implementation for each possible combination of known or unknown arguments and results (binding pattern) passed to the function. Binding patterns should not be mixed up with overloading of AMOS II functions [16].

In the implementation of the multi directional foreign functions it is possible to define cost and a selectivity functions associated to certain access paths. This way the query compiler will have an easier job translating the AMOSQL statement with the foreign function in it [4].

3.2.3 AMOSQL

AMOSQL is the declarative query language of AMOS II and can be described as a subset of the OO-parts of SQL 99. It is relationally complete. AMOSQL is based on OSQL and DAPLEX with functionality added for mediation, multi-directional foreign functions, late binding and active rules with operations applied over the AMOS II data model. The language is a combination of a Data Definition Language (DDL) and a Data Manipulation Language (DML). The syntax of AMOSQL is similar to the syntax used in SQL [4].

The semantics of an AMOSQL query can in general be described as [2]:

1. Form the Cartesian product of the type extents in the from-clause.

2. Restrict the Cartesian product by the condition.

3. For each possible variable binding to elements in the restricted Cartesian product, evaluate the result expression to form a result.

The fact that queries expressed in AMOSQL are declarative makes them dependent of the query optimiser for extensive optimisation before execution [25].

3.3 Extensibility

An important aspect in every software system is its ability in allowing for new functionality to be added making the system extendable [23]. In AMOS II this is, for example, implemented through a generalized foreign function mechanism providing access to special purpose data structures. A foreign function is defined in an external programming language. Currently AMOS II have interfaces to Java, C and Lisp [16].

3.3.1 Amos II Java interface

AMOS II has an interface to the object-oriented programming language Java. The communication between AMOS II and Java can be of two different types:

- A program written in Java calls AMOS II through the callin interface.

- AMOS II calls a Java program through the callout interface using foreign functions [16].

There is a possibility for the developer to use a combination of both interfaces writing programs that begin execution outside AMOS II and then during execution uses the callin interface to communicate with AMOS II [16].

The AMOS II Java API provides the developer with a number of methods to manage the communication between the application and AMOS II. Methods exist for accessing literal elements, executing queries, iterating through rows in a record set, creating/deleting objects and updating. The core of the API is made up of the following classes [16]:

- `Connection`. Represents a connection to an AMOS II database.
- `Scan`. A result set returned from the database consisting of tuples, or rows, of data.
- `Tuple`. Corresponds to a single row in the result set.
- `Oid`. Represents a corresponding AMOS II database object.

3.3.2 Java foreign functions

Foreign functions in Java are defined as methods of some user defined Java class stored in some external file with the same name as the class in order for AMOS II to find it. After the class has been compiled it can be dynamically loaded or linked into AMOS II by the creation of a foreign function [16].

A Java foreign function generally consists of three parts[16]:

1. The code to implement the function. A foreign function is implemented just like any other Java function with a signature and a body. The signature of a foreign function in Java has the following general syntax:

   ```
   public void <functionname>(CallContext <argname1>, Tuple <argname2>) throws
   AmosException;
   ```

2. A definition of the foreign function in AMOSQL. Hooking up the foreign function to the AMOS II system is managed with a special mechanism called a *resolvent* for the function. The resolvent tells the system how many arguments are sent to the function, their type, mutual order, and the type of the result. It also specifies the class containing the function, and the function's name. The resolvent is then assigned to a foreign function with the AMOSQL statement according to the following syntax [16]:

   ```
   create function <functionname>(<argument declaration>)->
   <result declaration>
   as foreign'JAVA:<class file>/<methodname>';
   ```

3. An optional *cost hint* to estimate the cost of executing the function. As the consequence of different implementations for multi-directional foreign functions one multi-directional foreign function can have different execution costs. A problem arise when the optimiser needs to choose between two possible execution plans for a query. In order for the query optimiser to choose the most efficient implementation cost hints can be used. Cost hints include the cost of executing the function and the *fanout* of the call for a given function where fanout is a measure of the expected size of the result of the function call [16].

4. The ORWIF project

ORWIF (Object-Relational Wrapper of Internet Forms) is an extension to the mediator database system AMOS II. ORWIF adds another feature to AMOS II's already existing wrapping mechanisms enabling the wrapping of Internet web forms. In essence ORWIF provides users of AMOS II with the possibility to combine data retrieved from different Internet web forms with data from regular databases and other web sources. This way a query in AMOSQL can be written so that it references different web forms.

ORWIF utilizes Internet form wrappers automatically generated by an Internet wrapper toolkit to extract information from web forms. These wrappers are hooked up to AMOS II with the help of foreign functions written in Java and accessed through the AMOS II Java interface. The program is started with a call to a foreign function supplying the necessary parameters.

Implementing the foreign functions in Java is suitable for several reasons: the application developed is not time critical, Java is relatively safe (the language does not provide any pointer mechanism), programmers can use libraries for common functions making it easier to implement the application [27] and finally many Internet wrapper toolkits often provide transparent access in Java to extracted information in form of objects.

4.1 Purpose

The purpose of the ORWIF project is that a user should be able to specify an AMOSQL query combining data from an AMOS II database or some other source of information with data retrieved from a chosen web form. This implies that the AMOS II functionality has to be extended so that web forms can be accessed directly from AMOSQL.

Through a user perspective this functionality facilitates the process of collecting, and analysing data on the Internet, e.g searching for a particular item, comparing features of items, etc.

4.2 Scenario

Today it is a time consuming task to search for and compare features of items sold by different vendors on the Internet. Therefore there is a need for a more efficient and faster way of searching and analysing information of items on the web. An example of such a search and analysis has been performed in this project.

Three web forms were to be picked from the Internet to be used in the ORWIF project. A couple of constraints should hold on the chosen web forms:

- They should conform to a certain level of complexity and therefore can not have fewer than three input fields.

- They were also supposed to be flexible meaning that the user should be able to give any value he or she wanted as form input values. Input values should also be volatile.

Finally web forms from the following web sites were chosen for the project:

- Amazon (htttp://www.amazon.com). An international Web site offering its visitors cheap prices on books, records, and a lot of other things.

- Ginza (http://www.ginza.se). A Swedish much smaller equivalent to Amazon.

- XE (http://www.XE.com). An international site for currency conversion. To be able to compare the prices on different records the XE site is used for automatic conversion of currency from American dollars on Amazon to Swedish kronas on Ginza.

In the scenario the goal is to provide an alternative much faster and easier way of collecting information regarding music from the web than by manually surfing these web pages. A user should be able to execute AMOSQL queries involving searches on Amazon, Ginza, and XE where data from these web sites are combined with data from an AMOS II embedded database. The search is executed through the submitting of a web form on each site and the resulting information are accessed through the wrapping of these web forms with the help of some Internet wrapper toolkit previously discussed in chapter 2.3.

Example of queries could be to find the name of a record with a special song on it, the name of a record with a special artist on it, how many records an artist has released, which artists that have performed together etc. Sending a '*' as an argument means that the value is omitted.

The following AMOSQL query tries to find an Elvis Presley recording of the song 'Jailhouse Rock", on vinyl at the Ginza web site:

```
select realname, title, atoi(price), currency, format
from charstring realname, charstring title, charstring price, charstring
currency, charstring format
where <realname,title,price,currency,format> =
Ginza_webform('Elvis','*','Jailhouse Rock','*','*') and
format = 'Vinyl';
```

The above AMOSQL query is quite simple and straightforward but queries can also be more complex. In the example below a derived function is first declared to find the cheapest album among several results.

Derived function:

```
create function album_price(charstring song) -> bag of number
as select price
from charstring art, charstring tit, charstring curr, charstring pri,
charstring form, number price
where <art,tit,pri,curr,form> = Ginza_webform('*','*',song,'*','*') and
price = atoi(pri);
```

The function is then used in the following AMOSQL query:

```
select art, tit, price, curr
from charstring art, charstring tit, charstring curr, charstring pri,
charstring form,number price
where <art,tit,pri,curr,form> = Ginza_webform('*','*','Jailhouse
Rock','*','*')
and price = atoi(pri)
and price = minagg(album_price('Jailhouse Rock'));
```

This query returns the artist name, title, and price of the cheapest album with a version of Jailhouse Rock on it when searching the Ginza web site through its web form.

A commercial application offering a similar service as ORWIF is the Price Runner web site where customers can log in to access the best price of capital goods before purchasing [26].

4.3 Requirements

In the process of developing ORWIF certain requirements has to be fulfilled by the programmer for the application to function in a satisfying way. The first two requirements concern the dynamic nature of Internet as mentioned in chapter 2.3.1.

- It must be possible to add a new wrapper class to the ORWIF package without being forced to recompile the whole project. The Java Virtual Machine then finds all wrapper classes automatically.

- Ease of adding new sources or modifying existing wrappers is of highest importance. New HTML wrappers will be added in the future and it is necessary that this procedure can be performed within a short period of time (at most one day). Also, people who are not that familiar with wrappers should be able to create a new wrapper within a few hours implying that the wrapper generator used cannot be too complicated .

The third requirement is about performance.

- Currently the only requirement regarding the performance of the application is about memory. The application should be designed to conform to an acceptable level of memory consumption when executed and not swamp the main-memory with complex data structures.

4.4 Choice of Internet wrapper toolkit

There are several wrapper toolkits available for the development of Internet wrappers today. The Internet wrapper toolkit finally chosen to be used in the ORWIF project was World Wide Web Wrapper Factory (W4F). The choice can be motivated as follows:

- W4F has declarative extraction rules. With declarative extraction rules the user does not have to focus on how to extract data. Instead he can put all his energy on deciding what data he

want to extract. Something that is often more important when the wrapper is part of a bigger application.

- W4F does not make itself dependent on old versions of required components. In this particular situation the JEDI wrapper generator was not thoroughly tested with the latest release of Sun's JDK 1.2 and had not been tested at all with JDK 1.3 The last JDK successfully tested with JEDI was version 1.1.6. However, it was not only JEDI who had problems with new versions of the Java environment. In order for WebL to work with JDK 1.3 some alterations had to be done in the WebL Java class `AutoStreamReader`.

- W4F quickly allows the user to start producing wrappers and has a straightforward way of integrating the code to Java applications. The concept to divide the development of your wrapper into several predefined sections gives you at an early stage a good overview of what you are doing and shortens the time it takes to create your first wrapper. W4F generates a Java class for every created wrapper making it transparent to integrate the wrapper with your Java application. Another feature of W4F is that the extraction rules can be obtained for free from the *wysiwyg* (what-you-see-is-what-you-get) interface.

- W4F has a very informative and instructive documentation. Sorry to say no program is self-explaining in how to use it. Some kind of information is needed. In my opinion the manuals from WebL and JEDI was all comprehensive and informative. The problem was that they were not instructive enough.

In the following chapter, W4F, together with two other acknowledged Internet wrapper toolkits, WebL and JEDI, are described in greater detail

4.5 Description and comparison of three Internet wrapper toolkits

There are several wrapper toolkits available for the development of Internet wrappers today. Most of them have facilities to extract both more structured information such as XML and less structured data in the form of HTML. Many of them can be downloaded from the web and are free for use on a non commercial basis though they sometimes demand the user to get a license first. In some cases the wrapper toolkit is still in its testing phase. This chapter describes three available wrapper toolkits and presents a comparison of them in table 1.

4.5.1 Compaq Web Language

Compaq Web Language (WebL) is based on the wrapper interpretator approach. Its web source wrappers are specified through a high level, object oriented, scripting language suitable for retrieving web pages, extracting and manipulating data from them. In essence the interpretated language is built up off *service combinators* and a *mark-up algebra* that are integrated in a small programming language core conceptually similar to most other procedural programming languages [12]. The WebL package also consists of a parser needed to validate and generate representations of fetched web pages.

In WebL, service combinators are language constructs providing the programmer with an opportunity to mimic the behaviour of a web surfer when a fail occurs while retrieving a web page. In essence the constructs makes predefined algorithmic behaviour scriptable like handling reloading of pages, retrying of requests, termination of requests taking to long, etc. The markup algebra allows for the extraction and manipulation of data from web pages with the help of algebraic operations on set of markup elements, so called *piece-sets*.

After retrieving and parsing the page a *piece* can be defined as a contiguous text region in a document, identified by the starting and the ending position of the region. We can imagine positions as indices that indicate a character offset in the page. Pieces within piece-sets may overlap, be nested, or may belong to different pages. However, unlike mathematical sets that do not impose a particular ordering on their elements, piece-sets are always in a canonical representation in which pieces are ordered according to their starting position, and then their ending position in the document. This allows iterating over pieces in a set in the sequence they appear in the document, and also to pick the *n*th occurrence of a pattern by indexing into the piece-set. There are two ways of creating piece-sets: A common way to create a piece-set is to search for all the HTML or XML elements with a specific name e.g a structured search. Another way to create a piece-set is to search for character patterns ,with an unstructured search or pattern search, ignoring the markup elements [12].

Figure 4 shows the wrapping features of WebL applied on a general model of a web computation. This model is divided into three phases: input face, processing phase and output phase. Web pages are first fetched with the help of service combinators and parsed with a markup parser into pieces and piece-sets. Then the markup algebra are used for extracting or searching for, data values from a page, computing on those values and page manipulation. After the processing phase the page is regenerated from its internal representation and stored back on the web.

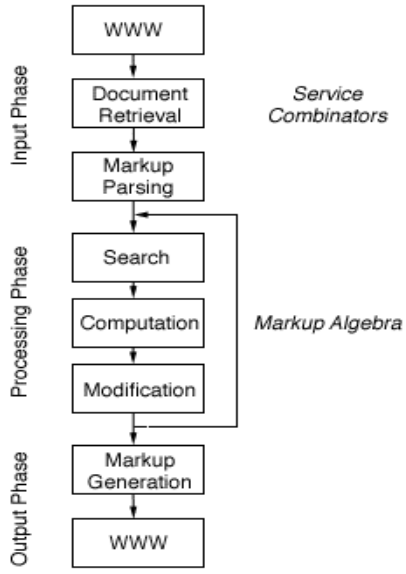


Figure 4: Web document processing with WebL [12].

The core functionalities of WebL consists of libraries written in Java implementing the parser, service combinators and markup algebra. The idea is that if a programmer don't want to use WebL syntax he should be able to use these libraries directly in his Java program. If he after all chooses to use the scripting language WebL offers a number of standard modules for extended functionality witch can be reused. One of them, the Java module, provide transparent access to any functionality provided by the Java class library meaning that a WebL internal object could easily be mapped to a Java object. Note that the direction of access is only from WebL to Java [12].

4.5.2 Java based Extraction and Dissemination of Information

Java based Extraction and Dissemination of Information (JEDI) is a light weight tool for creation of web source wrappers and mediators based on the wrapper interpretator approach. JEDI is entirely implemented in Java relying only on web browser technology and has facilities for extracting, combining, and reconciling data from several independent heterogeneous information sources.

The underlying data model of JEDI is self-descriptive and supports querying of structure and type of objects. Built in data-modelling facilities can be extended with Java classes to add new data types or to integrate data-models in a generic mode with the help of external libraries. Example of such data models are relational DBMS's, CORBA systems or document sources with a well defined format. Today JEDI has support for parsing and representing XML, HTML and RTF pages as DOM trees[13]. Wrappers that can be defined independently of the structure of data sources are called generic wrappers. Specific wrappers for sources with irregular and proprietary format need to take into account the structure of the source to be able to extract information [13].

Specific wrapping is enabled by the use of JEDI's scripting language and the JEDI parser. The scripting language is divided into two parts, *attributed grammar rules* and *control code*:

Attributed grammar rules describe source structure declaratively and are used in combination with pattern matching through regular expressions for managing extraction of data and assigning it to internal variables. The reason for using grammar rules is that pattern matching alone can not handle data in irregular sequences or data that is nested. Essentially a set of patterns can only describe the structure of a document as a flat set of objects. When the interpretation of patterns depends on their actual sequence or on their nesting structure patterns alone do not suffice.

Attributed grammar rules has a high expressive power and are accomplished by JEDI's fault tolerant parser allowing the JEDI scripting language to handle attributed, nested extraction rules but also incomplete and ambiguous source specifications. The latter are inevitable in order to define really robust extraction rules. When going trough the document JEDI always chooses the most specific extraction rule among several applicable ones, e.g the one witch produces the least overhead. If the parser doesn't find any applicable rule for some part of the document, instead of breaking, it skips as little of the document data as possible to find one that really is. This feature can be compared to existing grammar based parsing approaches avoiding or limiting ambiguity among extraction rules by imposing constraints on grammar specifications.

The other part of the JEDI information extraction language, the control code, provides functionality for object creation, calls to external functions or predicates, reading files, fetching web pages and parsing retrieved source data with the `parse` method [13].

The scripting language is very flexible and extensible as it allows Java classes and objects being easily accessed from JEDI. Also JEDI script objects can in a simple fashion be created from a Java application. JEDI can be run as a standalone application or be used from within an existing Java application. Mapping from JEDI variables to Java objects and the invertible should be straight forward. Further JEDI provides direct support (built into the scripting language) for DOM based document models which eases the task to create XML documents from semi-structured HTML sources. In other words it is very easy to map extracted data to XML element[13].

The use of regular expressions, and a fault tolerant parser makes JEDI to a very powerful, flexible and robust tool for generating wrappers suitable for a variety of application domains [13].

4.5.3 W4F

W4F is the wrapper toolkit used in the ORWIF project. It is a wrapper generator consisting of three parts: A *retrieval language* to identify Web sources, a *declarative extraction language* to express robust extraction rules and a *mapping interface* to export the extracted information into some user-defined data-structures. A HTML parser is also needed for formatting and validating the HTML tags. In order to facilitate and make the creation of wrappers rapid and easy, the toolkit offers some support in the form of ‘wizards’ permitting fast and semi-automatic generation of ready-to-go wrappers [7].

There are explicit rules describing how a W4F wrapper should operate when extracting information from web pages. These rules are expressed in a so called W4F description file with the suffix `.w4f`. The description file is compiled using the W4F compiler into Java classes that can be run as a stand-alone application using the W4F runtime. Code to be executed while being run in a stand alone mode, e.g a main method, is inserted in a special section of the description file. This implies the possibility to direct integration in Java applications and also convenient mapping between W4F data structures and Java objects [7].

Simplified W4F works as follows: An HTML document is first retrieved from the Web according to some retrieval rules. Once retrieved, it is handed over to a HTML parser constructing a representation, parse tree, following the Document Object Model standard. Extraction rules are then applied on the parse tree and the extracted information is stored in W4F’s flexible internal format *Nested List Strings* abbreviated NSL. An NSL is anonymous, it has no type and is a part of the mapping layer to map these anonymous structures into typed structures. Further, a NSL can handle any level of nesting and can be easily manipulated via a simple API. Finally, NSL structures could be mapped to the upper-level application, according to a given mapping scheme [14].

W4F uses the *HTML Extraction Language* (HEL) to extract information from HTML pages. HEL permits declarative specification of information extraction and is applicable to the W3C DOM model. The language allows description of extraction rules in terms of path-expressions along the

parse tree. There are two ways of navigating the tree. The first one follows the structure of the document implied by HTML tags. This type of navigation is extremely useful for searching table based document. The second one follows the flow of the document in an human like fashion. The two navigation styles complement each other and together they provide a way to identify most structures in terms of extraction paths. HEL also offers the capabilities to capture finer details of the document, such parts of text chunks, which can't be extracted with the help of pattern matching [14].

The information flow when using W4F to extract information from a HTML page is described by the figure below:

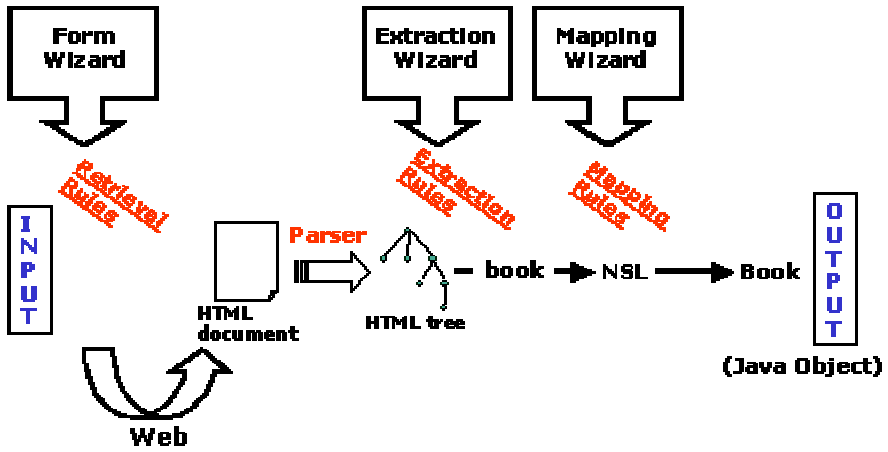


Figure 5: Information flow in W4F [28].

4.5.4 Comparison between three Internet wrapper toolkits

The table below is a comparison between the wrapper toolkits mentioned previously in chapter 4 over some chosen properties put together by the author:

Table 1: Comparison properties between three Internet wrapper toolkits.

Name:	WebL	JEDI	W4F
Type of wrapper toolkit:	wrapper-interpretator	wrapper-interpretator	wrapper-generator
URL:			
Source code available:	yes	no	no
Documentation:	available	available (for extracting XML)	available
Licence:	free for non-com use	free for non-com use	free for educational pur.
Treated Data Types:	plain text, HTML, XML	relational DBMS, CORBA, plain text, XML, HTML, RTF	HTML
Mapping to:	internal WebL objects,	internal JEDI objects, access to Java objects, XML element obj.	Nested String Lists, easy conversion to Java, XML.

Extraction rules:	non-declarative	declarative	declarative
Language:	scripting language	scripting language	no language
Required components:	JDK	JDK, DOM-implementation (HTML, XML,RTF), Swing1.1	JDK, DOM-impl. (HTML), javaregex
Supported protocols:	http, ftp	http,ftp	http, ftp
Level of automation:	no automation (only some example scripts)	no automation (only some example scripts)	semi-automated (generation of Java code)
Maturity of package:	no high-speed-front-end but a rapid prototyping tool.	available on the Internet for non-commercial use.	available on the Internet for non-commercial use
Characteristics:	interpretated wrappers with service-comb.	interpretated wrappers, very powerful and flexible with a unique parsing strategy.	generating a Java class for each wrapped page, robust and easy to use.

4.6 Architecture

ORWIF is built on top of the existing AMOS II Java interface and extends the architecture of AMOS II. It has multiple layer architecture, as shown in figure 6, fulfilling the requirements earlier mentioned in chapter 4.3. With ORWIF, the user can execute AMOSQL queries, referencing information from several web pages and combine this information with locally stored data or data collected from other sources.

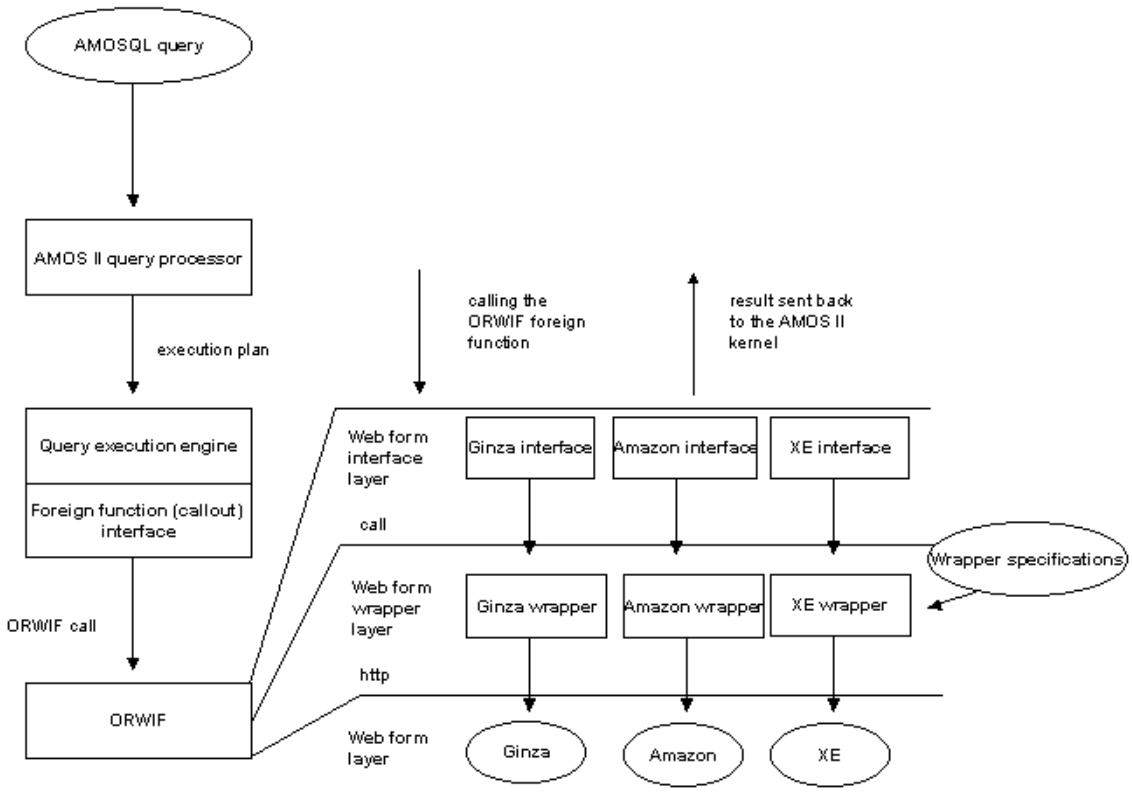


Figure 6: The ORWIF system architecture.

The left part of the figure show how ORWIF interface with the AMOS II kernel. ORWIF is executed with a call to a foreign function written in Java via the AMOS II Java callout interface.

The right part is a more detailed description of ORWIF itself. In the figure the web form interface layer defines an interface between the AMOS II kernel and the underlying web form wrapper layer. This is basically a collection of foreign functions used for communication between these two instances called by the query processor. Input is sent to the foreign function in the form of user defined parameters which are processed by the function and then sent to the underlying web form wrapper layer. The web form interface layer is also responsible for passing the data extracted by the underlying layer back to the AMOS II kernel. From the Amos II kernel's point of view the web form wrapper interface is simply the implementation of several foreign functions. Everything else is hidden below.

The web form wrapper layer consists of the modules specified through the Internet wrapper toolkit used. In this layer requests for the retrieval of data from a web form is formed and sent to the web server for execution through the http protocol. The web wrapper layer also extracts the wanted information from the resulting HTML page. For the wrapper toolkit to produce functioning modules it needs input in the form of specifications of submission and data extraction rules for a web source. In this case the chosen Internet wrapper toolkit, W4F, is a wrapper generator which generates Java classes for every wrapped data source. Therefore the layer only consists of generated code. Had the Internet wrapper toolkit used been an interpretator the layer would have consisted of the interpretator together with the extraction specifications.

Through the use of a multiple layer architecture it is possible to add a new wrapper class to the ORWIF package without being forced to recompile the whole project. The user simply adds a new web form wrapper and its belonging web form wrapper interface to the system. At execution time the newly added web form wrapper is automatically located by the Java Virtual Machine.

The procedure of adding new sources or modifying existing wrappers in ORWIF is also straightforward. The W4F wrapper toolkit offers its users a intuitive and simplistic approach for the developing of Internet wrappers so that people who are not that familiar with wrappers could create a new one within a fairly short period of time.

4.7 Implementation

ORWIF is implemented with AMOS II foreign functions, existing AMOS II Java libraries from the AMOS II Java API and an Internet wrapper toolkit, W4F.

The demo version of ORWIF consists of three foreign functions written in Java corresponding to three Java classes generated by the W4F wrapper-generator as showed in figure 7. The size of the compiled code is less than 50 kb. The ORWIF foreign functions was developed completely using Borland JBuilder 4 Professional Edition based on SUN's JDK 1.3.

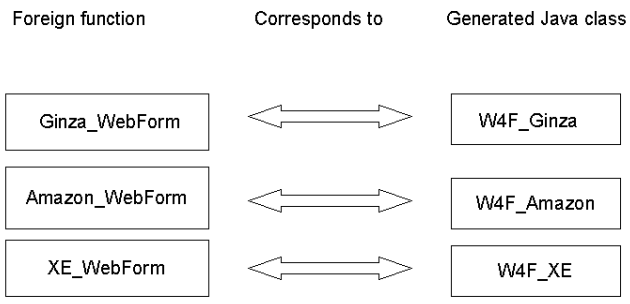


Figure 7: ORWIF foreign functions and corresponding Java classes

When a user executes an AMOSQL query referencing information from the Ginza web form this is what really happens: A static connection is set first in order to provide communication between the Amos II kernel and the Web form interface layer. This connection is established using foreign functions written in Java with the following signature:

```
public void Ginza_WebForm(CallContext cxt, Tuple tpl) throws AmosException
```

However before using the function it must be hooked up to the AMOS II system meaning that the function must be defined in AMOSQL. This is done by creating a resolvent for the function and then assigning this resolvent to a foreign function with an AMOSQL statement according to the following syntax:

```
create function Ginza_webform(charstring artist, charstring tit, charstring
song, charstring min, charstring max)
   -> <charstring name, charstring title, charstring price, charstring
currency, charstring format>
as foreign "JAVA:orwif.Ginza_FormWrapper/Ginza_WebForm";
```

After the function is called it first retrieves the arguments from AMOS II with the help of the `CallContext` and `Tuple` objects and assigns them to local variables. `CallContext` is used internally for managing the foreign function call and `Tuple` contains the arguments passed. The last position of `Tuple` represents a possible return value. Thus, if a foreign function has two arguments and one return value the `Tuple` object has three positions, indexed from zero to two.

What happens next is that the foreign function instantiates it's corresponding wrapper class, `W4F_Ginza`, in the web form wrapper layer through the static method `get.Ginza` and sends these local variables as arguments. The code implementing the procedure is shown below:

```
W4F_Ginza g = W4F_Ginza.getGinza(artist, title, song, price_range, kategori,
price, artist_type, title_type);
```

When the wrapper class has extracted the wanted information it is mapped to Java objects easily accessible from the foreign function according to mapping rules specified in the W4F description file. These objects can be reached from the foreign function with the `g` object handle.

In order to send the data back to AMOS II the foreign function now simply uses the earlier received `CallContext` object and a `Tuple` object. The reason why `CallContext` have to be the same is that this object is an identifier for a specific connection between AMOS II and the web form wrapper interface. AMOS II has to know that the results returned back from the foreign function applies to the right foreign function call. Objects of `Tuple` type on the other side can be used arbitrary. The result objects are placed in the previously defined `Tuple` object and are sent back to the AMOS II kernel. The result is then processed further by the AMOSQL query that originally started the operation and the final result is displayed on the screen.

To prevent the accumulation of non-used data in main memory ORWIF returns tuples instead of user defined objects as the latter are not automatically garbage collected by the system. This is an adequate solution when extensive querying in AMOSQL, referencing ORWIF foreign functions, after a period of time can produce a vast amount of objects.

Wrapping a web form means sending input parameters in a GET or POST request to the web server. Often a web form has it's own internal logic deciding witch patterns input values should follow for the web form to execute. Certain web forms do not execute unless several explicit non volatile input fields are provided with information.

If every input field of the web form would be specified by this internal logic mapping the web form to AMOS II foreign functions using binding patterns, thus producing multi directional foreign functions, should be a fairly trivial task. The programmer would know all the possible binding patterns needed to represent a specific web form. However, in most cases, only a small fragment of the web form is determined by this internal logic. The rest of the web form consists of optional input fields. In the present AMOS II representation of binding patterns there exist no such thing as optional parameters. Everything is either input or output.

To illustrate the problem it's a good idea to take a more closer look on the application of AMOS II foreign function binding patterns in the wrapping of web forms in real life:

Suppose a web form for searching music takes five optional input values in any order: song, price, artist, title, format. The most simple scenario is when all these input values has to be given in order to execute the web form. Mapping this and the following four successive scenarios to binding patterns produces the following table of known or input (-) and unknown or output (+) parameters:

Table 2: Mapping of input/output values to binding patterns

	Song	**Price**	**Artist**	**Title**	**Format**
First:	-	-	-	-	-
Second:	-	-	-	-	+
Third:	-	-	-	+	+
Fourth:	-	-	+	+	+
Fifth:	-	+	+	+	+
....					

To give a full representation of the web form we would have to consider $2^5 = 32$ binding patterns and corresponding foreign function definitions. Using a combination of binding patterns and overloading would reduce the numbers of binding patterns by 5 e.g $2^5 - 5 = 27$. Clearly, this is not really practical when such an amount of definitions has to be written to take care of all possible outcomes of a relatively simple web form with only 5 input fields.

If an optional operator had existed in the representation of AMOS II binding patterns a pattern could have been defined according to the table below.

Table 3: Mapping of input values to binding patterns with an optional operator

	Song	**Price**	**Artist**	**Title**	**Format**
First:	?	?	?	?	?

The binding pattern in table 3 carries enough information to cover all binding patterns represented in table 2. Also, instead of specifying 25 foreign function definitions corresponding to each binding pattern only one foreign function has to be specified

In ORWIF, binding patterns having an optional operator is simulated by using the '*'symbol when calling foreign functions. Sending a '*' means that the parameter value is omitted. The responsibility of handling such omitted parameter values is delegated to the foreign function being called. This is possible after adding necessary logic concerning parameter interpretation to the foreign function. For example it is important that a certain order is kept among the parameters sent to the foreign function or else it has no way of telling which value is left out.

All foreign functions in ORWIF has the capability to handle omitted parameter values. The following piece of code is an example of a foreign function call in ORWIF:

```
Ginza_webform('Elvis','*','Jailhouse Rock','*','*');
```

Table 4 gives an overview of the implementation of ORWIF based on the three web forms wrapped in this project:

Web form:	Ginza	Amazon	XE
Purpose:	Online media store	Online media store	Currency converter
Wrapper:	W4F_Ginza.w4f	W4F_Amazon.w4f	W4F_XE.w4f
Wrapper invoked with foreign-function:	Ginza_webform()	Amazon_webform()	XE_webform()
Possible arguments for foreign function:	Name of searched artist/group, title of record, title of song, min price interval, max price interval	Name of searched artist	Amount to change, from currency, to currency
Result from foreign function:	Name of artist/group, title, price, currency, format	Name of artist/group, title, price, currency, format	Result
Result type:	Tuple	Tuple	Tuple

Table 4: Overview of the implementation of ORWIF based on the wrapped web forms.

4.8 Adding a wrapper to ORWIF

This chapter is a summary of how to add a new web source to the AMOS II system based on what has been mentioned previously. The steps needed are the following:

1. Generate the wrapper with the wrapper package. There are both manual steps and automated steps involved in developing a wrapper. The manual steps consists of developing the W4F description file. A description file is as mentioned earlier divided into several sections. Each section starts with its section name followed by a "{" and ending with a "}". It has to be specified in the following order:

   The `OPTION` section allows the user to specify the DOM implementation to be used, timeout for retrieval of web pages etc.

   The `SCHEMA` section is where you specify how to map extracted information into Java objects.

   `EXTRACTION_RULES` and `RETRIEVAL_RULES` have already been described above.

   Finally, the `JAVACODE` section gives you the possibility to add your own Java code inside the `main` method. Actually the wrapper doesn't need any additional Java code to be used in the AMOS II system. The wrapper itself is not the main program. But to test the wrapper as a standalone application, it is necessary to use the `main` method.

   No one of the preceding sections is mandatory. Any combination off valid sections will produce some valid Java code. But your wrapper may not be very useful if you only define an `OPTIONS` section.

   The automated step in the creation of a wrapper is the generation of a java file from the description file and the compilation of the latter. This is simply done with the following piece of code:

   ```
   W4F <W4F source file>.w4f
   ```

   For each retrieval method a standard constructor to build an instance of your wrapper will be generated.

2. The second step is to write a foreign function implementation in Java. The function should among other things call a static method in the correlating web form wrapper class with the retrieved parameters from the foreign function call to instantiate the class. It should also handle the omitted parameter values, '*' , in a satisfying way.

3. Finally the generated wrapper should be hooked to AMOS by declaring a resolvent for the foreign function instantiating the wrapper.

5. Discussion

Information agents and mediator systems have been built to gather and integrate information from various sources. The mediator approach is used to integrate information from heterogeneous database systems, encapsulating the user from problems caused by different locations, different languages, different data representations and different protocols. An essential component of the mediator architecture is a wrapper around each individual data source, accepting queries from the mediator, translating the query into an appropriate format for the individual data source and finally returns the result to the mediator [9].

However, several problems arise when trying to access this data from the web compared to using conventional databases due to: the highly dynamic nature of web sources, the absence of a standardized interface, the lack of structure in web pages etc. Also, the shear number of information sources and that new sources frequently are added to the web imposes a huge problem to the developer of Internet wrappers.

Internet wrapper toolkits facilitates the development of Internet wrappers by the processing of wrapper specifications consisting of statements to connect to web sources and to detect the parts of the text to be extracted. They contribute with different ways to solve the problems mentioned earlier. Most of the Internet wrapper toolkits today include some advanced pattern matching language to extract data from Web documents. This way they are able to produce stable Internet wrappers that works even if the structure of the web source has slightly changed. One of the more interesting wrapper toolkits today is JEDI which uses a parser that is somewhat fault tolerant. This enables JEDI to cope with ambiguous and incomplete source specifications.

ORWIF is an example of an application with web wrapper facilities developed more specifically for the wrapping of web forms. ORWIF allows the user to do combined searches on web forms and locally stored data. An interesting feature of ORWIF is its simulation of the optional operator in binding patterns of foreign functions with the help of omitted parameter values. The responsibility of explicitly handling the omitted values sent as input to the function has been delegated to the function itself. The loss in performance caused by the solution in this case is negligible when ORWIF performance mainly is determined by networking factors.

In the process of developing ORWIF several problems aroused because of the specific nature of web forms. For example, the output format of the result page when submitting a form sometimes varies depending on the input parameters supplied. This problem is generally solved through manually checking all possible combinations of input parameters and the structure of the result page they produce. The foreign function and Internet wrapper specification for that particular web form is then developed in accordance to this information.

Another problem is the difficulty in accessing a web source with an Internet wrapper toolkit. In the wrapper specification of that specific web source there has to be defined which parameters are going to be sent to the server when submitting the web form. Submitting the web form with it's POST method makes it harder for the developer of the web wrapper to determine which parameters this is going to be. In the case of ORWIF a program named Net Cat was used to solve this problem. Net Cat basically provide its user with the functionality to start your own local server with an optional IP address. By choosing the IP address of the server called by the web form when submitted the user can be displayed the exact parameters being sent to the server.

Though sometimes, web forms use more sophisticated mechanisms in sending the parameters to the server, e.g cookies, usernames, passwords, additional http variables such as host or referer, to ensure that the web form is valid and is submitted the right way. In this case it is up to the Internet wrapper toolkit to provide sufficient functionality for these features.

When the use of script languages in the creation of web pages gets more frequent new demands are raised on the developers of web wrappers and on the toolkits they use. Internet wrapper toolkits has to be smarter and more flexible to deal with this new environment. Ideally they should be able to fully understand these scripting languages for them to be able to predict the behaviour and structure of a web page.

A lot of research is going on in the field of information gathering and integration from web sources. There is another extension of AMOS II called ORWIS (Object-Relational Wrapper of Internet Search Engines)[11] where multiple Internet search engines (ISE's), e.g Alta Vista or Google, are accessed instead of web forms. The system relies just like ORWIF on foreign functions written in Java and an Internet wrapper package, W4F, for facilitating the extraction of more or less structured data from web sources. At present, one major architectural difference between them is that ORWIF has an extra level of abstraction built into it, the OO ISE schema. This schema combined with the mediator facilities of AMOS II provides an extensible mechanism to express AMOSQL queries and OO views that combine data from several ISE's with data from other sources (e.g relational databases).

6. Conclusions and future work

The ORWIF project is an extension to AMOS II. It offers the user a flexible and easy way of accessing and analysing information from web forms combining it with data from other sources through an OO mediator database system. This functionality offers users to e.g do their own "price running" with ORWIF meaning that they have the possibility to access the best price of capital goods before purchasing it.

ORWIF was implemented using foreign functions written in Java and a publicly available Internet wrapper toolkit. The project has the following combination of features:

It is possible to plug in new web forms to the system.

Wrapping and adding the web form to the system is a simple task.

In this report I have addressed problems related to the retrieval and extraction of information from the web through web forms and how to integrate such information gathering facilities with the AMOS II system. The following results have been achieved:

- It has been described how the Amos II mediator architecture could be used when searching Internet web forms for information.

- In the process of choosing between different internet wrapper toolkits a description and comparison of three of them was presented: WebL, JEDI and W4F. Finally the W4F package was chosen. Not so much for its ability to produce extremely stable web wrappers, in that

case a more appropriate choice would have been the JEDI toolkit, but for its declarative extraction rules, its independency from old versions of components and its simplicity.

- An alternative solution to the problem of how to represent optional operators in binding patterns has been shown. The fact that AMOS II today lacks an optional operator for values in binding patterns that are unknown, led to that an amount of 27 foreign function definitions corresponding to each binding pattern should have to be written to take care of all possible outcomes of a relatively simple web form with 5 input fields. Instead a mechanism using omitted parameter values was developed to simulate binding patterns with an optional operator to reduce the overhead of foreign function definitions. The responsibility of handling the omitted values was delegated to the Java code in the foreign function called by the user.

ORWIF is in this stage only a prototype of a web form wrapper for AMOS II. In the future there are plans for building a translator module on top of the ORWIF wrapper. This would make it possible for the user to write queries to ORWIF on a more abstract OO level than before. The translator module should translate or rewrite this abstract OO query to lower level web form query specifications containing calls to their corresponding foreign functions.
In order to rewrite such high level OO queries the translator needs specific rewrite rules which must be supplied by the user or the administrator of the system.

Multithreading is another aspect discussed in conjunction with future developments of ORWIF. The wrapping of the HTML sources is still sequentially handled. Making the execution of ORWIF proceed in a multithreaded mode would improve the performance of the application.

Finally, there is a possibility hooking up an existing mp3 player to ORWIF allowing the user not only to see result of a search for a specific song but also to hear it. This could be done by calling the mp3 player from ORWIF via the AMOS II C interface. Such an addition to ORWIF functionality should be fairly simple to implement.

7. Example queries and results

Chapter 7 gives some examples of queries in AMOSQL referencing web forms through ORWIF and their typical execution times. These queries was performed during the month of march year 2001. Since then changes may have been made to the wrapped web form by the owner of the web site influencing the allowed input parameters/patterns or the structure of the result page. This implies that output and execution time of these queries below may not be exactly the same today as shown below.

1.Have John Coltrane and Archie Shepp performed together on an album and in that case how much does the record(s) cost?

The first one is built up of two derived functions, record by artist and perform_together?:

```
create function record_by_artist(charstring name) -> <charstring, charstring,
number, charstring>
as select all_artists, title, atoi(price), currency
from charstring all_artists, charstring title, charstring price, charstring
currency, charstring format
```

```
where <all_artists, title, price, currency, format> = Ginza_webform(name, '*',
'*', '*','*')
or <all_artists, title, price, currency, format> = Amazon_webform(name);

create function perform_together?(charstring artist1, charstring artist2)->
<charstring, charstring, number, charstring>
as select all_artists, title, price, currency
from charstring all_artists, charstring title, number price, charstring
currency
where <all_artists, title, price, currency> = record_by_artist(artist1)
and contains(all_artists, artist2);
```

It is executed with a call to the derived function, perform_together?:

```
perform_together?('John Coltrane', 'Archie Shepp');
```

Executing the query produces the following result:

```
<"Coltrane John/Archie Shepp", "New thing at Newport", 149, "kr">
```

The average execution time is about 7 sec.

2.What is the lowest possible price in dollars for the record "Back to earth" from Lisa Ekdahl on CD?

The second one is built up of one stored function, standard_currency_name, and two derived functions, convert_price and price_finder:

```
create function standard_currency_name(charstring) -> charstring;
add standard_currency_name('$') = 'USD';
add standard_currency_name('USD') = 'USD';
add standard_currency_name('kr') = 'SEK';
add standard_currency_name('SEK') = 'SEK';

create function convert_price(number price, charstring fr, charstring t) ->
number
as select atoi(new_price)
from charstring new_price, charstring old_price
where old_price = itoa(price)
and new_price = XE_webform(old_price, standard_currency_name(fr),
standard_currency_name(t));

create function price_finder(charstring name, charstring title)
-> <charstring, number, charstring>
as select realtitle, convert_price(atoi(realprice), realcurr, 'USD'), realcurr
from charstring realname, charstring realtitle, charstring realprice,
charstring realcurr, charstring realformat
where <realname, realtitle, realprice, realcurr, realformat> =
Ginza_webform(name, title, '*', '*', '*')
or <realname, realtitle, realprice, realcurr, realformat> =
Amazon_webform(name)
and contains(realtitle, title);
```

It is executed with a call to the derived function, price_finder:

```
price_finder('Lisa Ekdahl', 'Back to earth');
```

Executing the query produces the following result:

```
<”Back to earth 1998”, 7.45, ”kr”>
<”Back to earth”, 13.99, ”$”>
```

The average execution time is about 8 sec.

3. How many records has Frank Sinatra ever recorded?

The third query is built up of two derived functions, record_finder and record_count:

```
create function record_finder(charstring name) -> charstring
as select distinct title
from charstring realname, charstring title, charstring price,charstring curr,
charstring format
where <realname, title, price, curr, format> = Ginza_webform(name, '*', '*',
'*','*')
or <realname, title, price, curr,  format> = Amazon_webform(name);

create function record_count(charstring name) -> integer
as select c
from integer c
where c = count (record_finder(name));
```

It is executed with a call to the derived function, record_finder:

```
record_count(‘Frank Sinatra’);
```

Executing the query produces the following result:

```
45
```

The average execution time is about 7 sec.

Though the preceding queries are quite complex and CPU intensive during execution even now performance is reasonable. Noticeable is that all queries took much longer to answer the conventional way by manually surfing the web with a HTML browser.

References

[1] G Fahl, T Risch: AMOS II Introduction, UDBL, Uppsala University, Sweden, 1999

[2] T Risch, V Josivofski, T Katchaunov: "AMOS II Concepts", UDBL, Uppsala Universitet, Sweden, *http://www.dis.uu.se/~udbl/amos/doc/amos_concepts.html*, 2000

[3] S Flodin, V Josifovski, T Katchaounov, T Risch, M Sköld, M Werner: "AMOS II User's Manual", UDBL, Uppsala University, Sweden, *http://www.dis.uu.se/~udbl/amos/doc/amos_users_guide.html*, 2000

[4] T Risch, V Josifovski: "Distributed Mediation by Object-Oriented Mediator Servers", To be published in "*Concurrency – Practice and Experience*", J Wiley & Sons, *http://www.dis.uu.se/~udbl/pobl/concur00.pdf*, 2001

[5] G Wiederhold: "Mediators in the Architecture of Future Information Systems", *IEEE Cpmputer*, 25(3), 38-49, 1992

[6] J Melton, J Michels, V Josifovski, K Kulkarni, P Schwarz, K Zeidenstein: "SQL and Management of External Data", *SIGMOD Record*, Vol. 30, No. 1, March 2001

[7] W4F(World Wide Web Wrapper Factory): "Building light-weight wrappers for legacy web data sources using W4F", *Conf. On Very Large Databases (VLDB'99)*: 738-741,1999.

[8] A Firat, S Madnick, M Siegel: "The Caméléon Web Wrapper Engine", *First Workshop on Technologies for E-services*, Cairo, 2000

[9] N Ashish , C A Knoblock: "Semi-Automatic Wrapper Generation for Internet Information Sources", *Conference on Cooperative Information Systems*:160-169, 1997

[10] L Liu, W Han, D Buttler, C Pu, W Tang: XWRAP: "An XML - based Wrapper Generator for Web Information Extraction", *SIGMOD Conference*: 540-543, 1999

[11] T Katchaounov, T Risch, S Zurcher: "Object-Oriented Mediator Queries to Internet Search Engines", UDBL, Uppsala Universitet, 2001

[12] T Kistler, H Marais: "WebL - a programming language for the Web", *WWW7*, Brisbane Australia, 1998, http://research.digital.com/SRC/WebL/

[13] G Huck, P Franhauser, K Aberer, E J Neuhold: "JEDI: Extracting and Synthesizing Information from the Web", *CoopIS'98 Conference*: 32-43, 1998

[14] F Azavant, A Sahuguet: "World Wide Web Wrapper Factory (W4F) User Manual", University of Pennsylvania, USA, 2000

[15] Centre for Objekt Teknology (COT): "Database Management Systems: Relational, Object-Relational, and Object-Oriented Data Models", *COT/4-02-V1.1*, Denmark, 1998

[16] D Elin, T Risch: “AMOS II Java Interface”, UDBL, Uppsala University, Sweden, August 2000

[17] T Risch: “AMOS II External Interfaces”, UDBL, Uppsala University, Sweden, February 2000

[18] D Quass, A Rajaraman, Y Sagiv, J D Ullman, J Widom: “Querying Semistructured Heterogeneous Information”, In *Deductive and Object-Oriented Databases, Proceedings of the DOOD’95 conference* , LNCS Vol. 1013, 319-344, Springer 1995

[19] L Eikvil: “Information Extraction from World Wide Web-A Survey”, Norwegian Computing Center, Oslo, *http://citeseer.nj.nec.com/eikvil99information.html*, 1999

[20] R Goldman, J Widom: WSQ/DSQ: “A Pratical Approach for Combined Querying of Databases and the Web”, *SIGMOD 2000 Conference*: 32-43, 1998

[21] V Josifovski, T Risch: “Comparison of AMOS II with Other Data Integration Projects”, *Technical Report*, EDSLAB, Linköping University, *http://www.ida.liu.se/~edslab/amosII_comp.pdf*, 1999

[22] W3C, “The Document Object Model”, *http://www.w3.org/DOM*, 1998

[23] Elmasri, R, Navathe, S.B: “Fundamentals of database systems”, Addison-Wesley, USA, 1997

[24] McClure, S: “Object Database vs. Object-Relational Databases”, *IDC Bulletin #14821E*, 1997

[25] Risch, T, Josifovski, V: ”Distributed Mediation using a Light-Weight OODBMS” , In *1st ECOOP Workshop on Object-Oriented Databases*, Lisbon Portugal, June 1999

[26] http://se.pricerunner.com/

[27] B Eckel: “Thinking in Java”, Prentice-Hall, USA, 2000

[28] F Azavant, A Sahuguet: “W4F World Wide Web Wrapper Factory Overview”, Database Research Group, University of Pennsylvania, *http://db.cis.upenn.edu/W4F/overview.html*, 1999

[29] Hubert Naacke, Olga Kapitskaia, Antony Tomasic, Philippe Bonnet, Louiqa Raschid, Remy Amouroux: "The Distributed Information Search Component (Disco) and the World Wide Web" , *Proc. of ACM SIGMOD Conf. on Management of Data* , 1997

[30] M Roth, P Schwartz: “Don’t Scrap It, Wrap It”, *23th. Int. Conf. On Very Large Databases (VLDB’97)*, pp.266-275, Athens Greece, 1997.

[31] U, Dayal: “Processing Queries Over Generalization Hierarchies in a Multidatabase system”, *9th Conf. On Very Large Database Systems(VLDB’83)*, Florence Italy, 1983

Appendix

AMOSQL queries, results and response time in demonstration of ORWIF:

1. Is there any Elvis records on vinyl with the song 'Love me tender'?
   (Only Ginza.se)

```
create function song_search (charstring name, charstring song, charstring
format)
       -> <charstring, charstring, charstring, charstring, charstring>
as
select realname, title, price, currency, format
from charstring realname, charstring title, charstring price, charstring
currency, charstring f
where <realname,title,price,currency,f> =
Ginza_webform(name,'*',song,'*','*') and format = f;

song_search('Elvis', 'Love me tender', 'Vinyl');

<"Presley Elvis","Artist of the century 5-LP","199","kr","Vinyl">
```

Elapsed time:1 sek

2. Who has recorded the song 'Jailhouse Rock' and what is the cheapest
   album with a version of that song? This query is only executable at Ginza.

```
create function album_price(charstring song) -> bag of number
as select price
   from charstring art, charstring tit, charstring curr, charstring pri,
charstring form, number price
   where <art,tit,pri,curr,form> = Ginza_webform('*','*',song,'*','*') and
         price = atoi(pri);

create function cheapest_album(charstring s) -> <charstring, charstring,
number>
as select art, tit, price
from charstring art, charstring tit, charstring curr, charstring pri,
charstring form,
     charstring s, number price
where <art,tit,pri,curr,form> = Ginza_webform('*','*',s,'*','*') and
      price = atoi(pri) and
      price = minagg(album_price(s));

cheapest_album('Jailhouse Rock');

<"Jones Tom","Elvis Beatles & me","29","kr","CD">
<"Jordanaires","With friends","29","kr","CD">
<"Jordanaires/Danny Mirror","Elvis partytime","29","kr","CD">
```

Elapsed time: 28 sek

3. Have John Coltrane and Archie Shepp performed together on an album
   and in that case how much does the record(s) cost?

```
create function record_by_artist(charstring name) -> <charstring,
charstring, number, charstring>
as select all_artists, title, atoi(price), currency
   from charstring all_artists, charstring title, charstring price,
charstring currency, charstring format
   where <all_artists, title, price, currency, format> =
Ginza_webform(name, '*', '*', '*','*') or
         <all_artists, title, price, currency, format> =
Amazon_webform(name);

create function perform_together?(charstring artist1, charstring artist2)
       -> <charstring, charstring, number, charstring>
as select all_artists, title, price, currency
   from charstring all_artists, charstring title, number price, charstring
currency
   where <all_artists, title, price, currency> = record_by_artist(artist1)
and
         contains(all_artists, artist2);

record_by_artist('John Coltrane');
record_by_artist('Archie Shepp');
perform_together?('John Coltrane', 'Archie Shepp');
perform_together?('John Coltrane', 'Duke
Ellington');perform_together?('John Coltrane', 'Duke Ellington');

<"Coltrane John/Archie Shepp","New thing at Newport",149,"kr">
```

Elapsed time: 7 sek

4. What is the lowest possible price in dollars for the record
"Back to earth" from Lisa Ekdahl on CD?

```
create function standard_currency_name(charstring) -> charstring;
add standard_currency_name('$') = 'USD';
add standard_currency_name('USD') = 'USD';
add standard_currency_name('kr') = 'SEK';
add standard_currency_name('SEK') = 'SEK';

create function convert_price(number price, charstring fr, charstring t) ->
number
as select atoi(new_price)
   from charstring new_price, charstring old_price
   where old_price = itoa(price) and
         new_price = XE_webform(old_price, standard_currency_name(fr),
standard_currency_name(t));

create function price_finder(charstring name, charstring title) ->
<charstring, number, charstring>
as select realtitle, convert_price(atoi(realprice), realcurr, 'USD'),
realcurr
   from charstring realname, charstring realtitle, charstring realprice,
        charstring realcurr, charstring realformat
   where <realname, realtitle, realprice, realcurr, realformat> =
Ginza_webform(name, title, '*', '*', '*') or
         <realname, realtitle, realprice, realcurr, realformat> =
Amazon_webform(name) and
         contains(realtitle, title);

price_finder('Lisa Ekdahl', 'Back to earth');
convert_price(123, 'kr', '$');
convert_price(123, '$', 'kr');
```

```
convert_price(123, '$', '$');
convert_price(123, 'SEK', 'USD');

<"Back to earth 1998","7.45","kr"> (kronor anges som enhet för att visa på
att skivan finns på Ginza men är konverterade till dollar)
<"Back to earth","13.99","$">
```

Elapsed time: 8 sek

5. How many records has Frank Sinatra ever recorded according to Ginza and Amazon?

```
create function record_finder(charstring name) -> charstring
as select distinct title
   from charstring realname, charstring title, charstring price,charstring
curr, charstring format
   where <realname, title, price, curr, format> = Ginza_webform(name, '*',
'*', '*','*') or
         <realname, title, price, curr,  format> = Amazon_webform(name);

create function record_count(charstring name) -> integer
as select c
   from integer c
   where c = count (record_finder(name));

record_count("Sinatra");

45
```

Elapsed time: 7 sek

This code creates the ORWIF.dmp file:

```
create function contains(character str, character substr)->boolean
  as select like_i(str, "*" + substr + "*");

create function Ginza_webform(charstring artist, charstring tit, charstring
song, charstring min, charstring max)
           -> <charstring name, charstring title, charstring price,
charstring currency, charstring format>
as foreign "JAVA:orwif.Ginza_FormWrapper/Ginza_WebForm";

create function XE_webform(charstring amount, charstring fr, charstring t)
           -> charstring res
as foreign "JAVA:orwif.XE_FormWrapper/XE_WebForm";

create function Amazon_webform(charstring artist)
           -> <charstring name, charstring title, charstring price,
charstring currency, charstring format>
as foreign "JAVA:orwif.Amazon_FormWrapper/Amazon_WebForm";

save "../bin/orwif.dmp";
quit;
```

Java classes implementing the web form interface layer.

Amazon_FromWrapper:

```
package orwif;


import callin.*;
import callout.*;
import orwif.*;

public class Amazon_FormWrapper {

  String index = "music";
  String field_keywords;

    public void Amazon_WebForm(CallContext cxt, Tuple tpl) throws
AmosException {

        try {
            if (tpl.getStringElem(0).equals("*")){
              field_keywords = "";
            }else{
              field_keywords = tpl.getStringElem(0);}
        } catch(Exception e) {
          System.out.println(e);
      }


        try {
            W4F_Amazon a = W4F_Amazon.getAmazon(index, field_keywords);

            if (a.artist.length != 0){
                      System.out.println(a.artist.length);
                      System.out.println(a.artist[0]);
                      System.out.println("HEJSAN");
              for (int i = 0; i < (a.artist.length); i++) {
                  tpl.setElem(1,a.artist[i]);
                  tpl.setElem(2,a.title[i]);
                  tpl.setElem(3,a.price[i]);
                  tpl.setElem(4,"$");
                  tpl.setElem(5,a.format[i]);
                  cxt.emit(tpl);
              }

            }

        } catch (Exception e) {
              System.out.println(e.getMessage());
          }
    }
}
```

Ginza_FormWrapper:

```
package orwif;

import callin.*;
import callout.*;
import orwif.*;
import java.lang.*;
import java.util.*;

public class Ginza_FormWrapper {

  String artist;
  String title;
  String song;
  String price_range;

    public void Ginza_WebForm(CallContext cxt, Tuple tpl) throws
AmosException {

        String artist_type="2";
        String title_type="2";
        String kategori="0";
        String price="0";

            try {
                if (tpl.getStringElem(0).equals("*")){
                    artist = "";
                }else{
                    artist = tpl.getStringElem(0);}
                if (tpl.getStringElem(1).equals("*")){
                    title = "";
                }else{
                    title = tpl.getStringElem(1);}
                if (tpl.getStringElem(2).equals("*")){
                    song = "";
                }else{
                    song = tpl.getStringElem(2);}
                if
(tpl.getStringElem(3).equals("*")||tpl.getStringElem(4).equals("*")){
                    price_range = "";
                }else{
                     price_range = (tpl.getStringElem(3)+"-
"+tpl.getStringElem(4));}

            } catch(Exception e) {
              System.out.println(e);
              }

            try{
                W4F_Ginza g = W4F_Ginza.getGinza(artist, title, song,
price_range, kategori, price, artist_type, title_type);

                if(g.title.length != 0){
                  for (int i = 0; i < (g.title.length); i++) {
                    if(g.name[i] != null){
                      tpl.setElem(5,g.name[i]);
                    }else{
                      tpl.setElem(5,"No such data");}
                      tpl.setElem(6,g.title[i]);
```

```
                    tpl.setElem(7,g.price[i]);
                    tpl.setElem(8,"kr");
                    tpl.setElem(9,g.format[i]);
                    cxt.emit(tpl);
                }

            }

        } catch (Exception e) {
            System.out.println(e.getMessage());
          }
    }

}
```

XE_FormWrapper:

```
package orwif;

import callin.*;
import callout.*;
import orwif.*;

public class XE_FormWrapper {

  String cur;
  String fr;
  String to;

    public void XE_WebForm(CallContext cxt, Tuple tpl) throws AmosException
{

        try{
            if (tpl.getStringElem(0).equals("*")){
                cur = "";
            }else {
                cur = tpl.getStringElem(0);}
            if (tpl.getStringElem(1).equals("*")){
                fr = "";
            }else {
                fr = tpl.getStringElem(1);}
            if(tpl.getStringElem(2).equals("*")){
                to = "";
            }else {
                to = tpl.getStringElem(2);}
        } catch(Exception e){
              System.out.println(e);
      }



        try {

            W4F_XE x = W4F_XE.getXE(cur, fr, to);

            if (x.result != null){
                tpl.setElem(3, x.result);
                cxt.emit(tpl);
            }
```

```
        } catch (Exception e) {
              System.out.println(e.getMessage());
          }
    }}
```

Wrapper specification files.

W4F_Amazon.w4f:

```
OPTIONS {

           package orwif;


}

SCHEMA {
           String artist[];
           String title[];
           String price[];
           String format[];

}

EXTRACTION_RULES
{

           artist = html.body[0].table[2-
].tr[t:*].td[1].table[0].tr[0].td[2].font[0].font[0].pcdata[0].txt,
split("~"), flatten(2)
where  html.body[0].table[2-
].tr[t].td[1].table[0].tr[0].td[2].font[0].font[0].pcdata[0].txt != null;

           title = html.body[0].table[2-
].tr[i:*].td[1].table[0].tr[0].td[2].font[0].b[0].a[0].pcdata[0].txt,
flatten()
where html.body[0].table[2-
].tr[i].td[1].table[0].tr[0].td[2].font[0].b[0].a[0].pcdata[0].txt != null;

           price = html.body[0].table[2-
].tr[y:*].td[1].table[0].tr[1].td[0].table[0].tr[0].td[0].font[0].b[0].font
[0].pcdata[0].txt, match("([0-9]+.*?[0-9]+)"), flatten() where
html.body[0].table[2-
].tr[y:*].td[1].table[0].tr[1].td[0].table[0].tr[0].td[0].font[0].b[0].font
[0].pcdata[0].txt != null
and html.body[0].table[2-
].tr[y:*].td[1].table[0].tr[1].td[0].table[0].tr[0].td[0].font[0].b[0].font
[0].pcdata[0].txt =~ "[0-9]+";

           format = html.body[0].table[2-
].tr[f:*].td[1].table[0].tr[0].td[2].font[0].pcdata[0].txt, match("(Audio
CD)"), flatten(2)
where  html.body[0].table[2-
].tr[f].td[1].table[0].tr[0].td[2].font[0].pcdata[0].txt != null;

}


RETRIEVAL_RULES
```

```
{
           getAmazon(String s1,String s2){
                      METHOD: POST;
                      URL: "http://www.amazon.com/exec/obidos/search-
handle-form/102-1885851-8366540";
                      PARAM:     "index"="$s1$",
                                 "field-keywords"="$s2$";
           }



}
JAVA_CODE
{
           public static void main(String args[]) throws Exception {
                      W4F_Amazon a =
W4F_Amazon.getAmazon("music","Presley");
                      System.out.println(a);
           }
}
```

W4F_Ginza.w4f:

```
OPTIONS {

           package orwif;

}

SCHEMA {
           String name[];
           String title[];
           String price[];
           String format[];
}

EXTRACTION_RULES
{
           name = html.body[0].table[1].tr[i:*].td[0].b[0].pcdata[0].txt
           /*where html.body[0].table[1].tr[i].td[0].b[0].pcdata[0].txt !=
null*/
           /*The variable 'name' must be allowed to tahe the value null
since every record doesn't have an artist*/
           where html.body[0].table[1].tr[i].td[0].b[0].pcdata[0].txt !=
"Artist/Grupp"
           and html.body[0].table[1].tr[i].td[0].pcdata[0].txt !=~
"Resultat";

           title =
html.body[0].table[1].tr[i:*].td[1].b[0].a[0].pcdata[0].txt
           where html.body[0].table[1].tr[i].td[1].b[0].a[0].pcdata[0].txt
!= null;

           price = html.body[0].table[1].tr[i:*].td[3].pcdata[0].txt,
match("([0-9]+)")
           where html.body[0].table[1].tr[i].td[3].pcdata[0].txt != null
           and html.body[0].table[1].tr[i].td[3].pcdata[0].txt =~ "[0-9]+";

           format = html.body[0].table[1].tr[i:*].td[2].pcdata[0].txt
           where html.body[0].table[1].tr[i].td[2].pcdata[0].txt != null
```

```
           and html.body[0].table[1].tr[i:*].td[2].pcdata[0].txt !=
"Kategori";


}


RETRIEVAL_RULES
{
           getGinza(String s1, String s2, String s3, String s4, String s5,
String s6, String s7, String s8){
                      METHOD: GET ;
                      URL:
"http://www.ginza.se/search_advanced.asp?artist=$s1$&titel=$s2$&lat=$s3$&fr
ipris=$s4$&kategori=$s5$&pris=$s6$&artist_type=$s7$&titel_type=$s8$";
           }

}

JAVA_CODE
{
           public static void main(String args[]) throws Exception {
                      W4F_Ginza g =
W4F_Ginza.getGinza("","song","","","0","0","2","2");
                      System.out.println(g);
           }
}
```

W4F_XE.w4f:

```
OPTIONS {

           package orwif;

}

SCHEMA {

           String result;

}

EXTRACTION_RULES
{

           result =
html.body[0].font[0].table[1].tr[0].td[0].table[0].tr[1].td[0].table[0].tr[
1].td[0].table[0].tr[0].td[4].font[0].font[0].b[0].pcdata[0].txt, match
("(.*?) ");



}


RETRIEVAL_RULES
{
           getXE(String cur, String fr, String to){
                      METHOD: POST;
                      URL: "http://www.xe.net/ucc/convert.cgi";
```

```
                    PARAM:      "Amount" = $cur$,
                                "From" = $fr$,
                                "To" = $to$;


        }

}

JAVA_CODE
{
        public static void main(String args[]) throws Exception {
                    W4F_XE x  = W4F_XE.getXE("10","USD","SEK");
                    System.out.println(x);
        }
}
```