Paper number :	**EUROPE64**
Title :	**Active Views for Electronic Commerce**
Contact :	Tova, Milo
Email :	milo@math.tau.ac.il
Additional Authors :	Serge, Abiteboul, abitebou@cosmos.inria.fr
	Bernd, Amann, amann@cosmos.inria.fr,
	Sophie, Cluet, cluet@cosmos.inria.fr,
	Anat, Eyal, aeyal@mekorot.co.il,
	Laurent, Mignet, mignet@cosmos.inria.fr

Active Views for Electronic Commerce*

S. Abiteboul†
I.N.R.I.A
Rocquencourt, France

B. Amann‡
Cedric/CNAM
Paris, France

S. Cluet†
I.N.R.I.A
Rocquencourt, France

A. Eyal§
U. Tel Aviv,
Tel Aviv, Israel

L. Mignet†
I.N.R.I.A
Rocquencourt, France

T. Milo¶
U. Tel Aviv,
Tel Aviv, Israel

February 22, 1999

Abstract

Electronic commerce is emerging as a major Web-supported application. In this paper we argue that database technology can, and should, provide the backbone for a wide range of such applications. More precisely, we present here the ActiveViews system, which, relaying on an extensive use of database features including views, active rules (triggers), and enhanced mechanisms for notification, access control and logging/tracing of users activities, provides the needed basis for electronic commerce.

Based on the emerging XML standards (DOM, query languages for XML, etc.), the system offers a novel declarative view specification language, describing the relevant data and activities of all actors (e.g. vendors and clients) participating in electronic commerce activities . Then, acting as an application generator, the system generates an actual, possibly customized, Web application that allows users to perform the given set of controlled activities and to work interactively on the specified data in a standard distributed environment.

Although closely related to workflow management systems, a major difference here is the importance we give to data. While workflow systems give declarative means for specifying the operations flow, the data involved is typically described in a very abstract manner, often disconnected from the description of the flow itself. In contrast, our approach promotes the data, acknowledging its importance for optimization coherence and analysis of applications.

The ActiveView system is developed at INRIA on top of Java and ArdentSoftware's XML repository.

1 Introduction

Internet has revolutionized the electronic publication of data. We should expect to see more and more Internet applications allowing clients to interact on the net notably by sharing data. It is possible to develop such applications today but this is at the cost of intense software developments by sophisticated programmers. We believe that (i) the need for fast application deployment, (ii) the generalization of such applications, and (iii) the often-met requirement of *proving* properties of these applications, will require the use of declarative specifications of applications. The situation is somewhat similar to what lead in the 70's to declarative query

*Work partially founded by a French Israeli grant.

†Email: <firstname>.<lastname>@inria.fr
‡Email: amann@cnam.fr
§Email: aeyal@mekorot.co.il
¶Email: milo@math.tau.ac.il

languages. Indeed, we believe that declarative query languages and databases form an essential component of the problem. This paper proposes such a specification language (the *ActiveView* language) and discusses how it is supported in the *ActiveView* system.

To illustrate the issues, consider electronic commerce. (Our examples will be based on a Web catalog.) Electronic commerce is emerging as a major Web-supported application. In a nutshell, electronic commerce supports business transactions between multiple parties via the network. This activity has many aspects, including security, authentication, electronic payment, and designing business models [25]. Electronic commerce also requires database support, since it often involves handling large amounts of data (e.g. product catalogs, yellow pages, etc.) and must provide transactions, concurrency control, distribution and recovery. It also involves strong interactions between participants (e.g., customers and vendors) and a control of the sequencing of activities (i.e., workflow management). All these aspects will be addressed by active views.

More generally, the applications we are interested in involve: (i) sharing of data and (ii) some cooperative work by a number of actors connected via the network. These are typical features found also for instance in digital libraries or information manufacturing systems.

We believe that database technology provides the backbone for such applications. Indeed, the ActiveView system can be seen as a *database application generator*. The system enables a *declarative* specification of *certain kinds* of database applications. By declarative, we mean here that there is little (or no program) to write and that the description of the application is in a high level language (or via a graphical user interface). The specification of an application includes definitions of the main actors involved in the application. For each actor, we specify:

1. the data and operations available to this particular actor (a *view* mechanism) and these with a sophisticate access control;

2. the activities this actor may be engaged in and the data and operations available in each;

3. some active rules that notably specify the sequencing of activities (a *workflow* component) but also the events this actor wants to be notified of (a *subscription* component) and those that have to be logged (a *tracing* component).

So, the ActiveView language allows to declaratively specify a number of features that are often considered in isolation. A main contribution of this paper is to show how these various aspects may be combined in a simple coherent framework. Active views rely heavily on four key components:

1. *XML :* From a data viewpoint, we selected the eXtended Markup Language (XML) [20] as the model for data.[1] All data stored, exchanged or presented to users are XML.

2. *Active rules :* Our active rules are rather simple compared to what may be found in the literature [24, 15]. The novelty is in the way they are integrated into a general framework and the way they are used for many purposes (workflow, change control, tracing).

3. *Method calls and notifications :* The events that enable active rules are method calls. The system rely on some subscription mechanism that allows views to be notified of certain events.

4. *View management :* Views have been quite studied in databases [9, 11]. We build here on our experience with O_2-Views [17], a system developed at INRIA. The views we are considering here are much simpler. The novelty is in the combination with active features.

[1]The Web has so far relied primarily on HTML that emphasizes an hypertext document approach. XML, although originally a document mark-up language, includes more structure. It is believed that XML will soon be the standard for data exchanges on the Web.

To see an example, suppose a product is added to the catalog. A notification is issued to all actors that are interested in this event, i.e. a change in the catalog. For instance, vendors may want to always see the most recent version of the catalog. Their specification should thus include an active rule to specify that, when such an event occurs, their view of the catalog should be updated. Observe that both the detection of the event and the maintenance may take advantage of incremental techniques. In particular, if the update affects a portion of the catalog a specific vendor is not interested in, we should avoid updating the view. Furthermore, when a vendor view has to be updated, we want to do it incrementally to avoid re-sending large portions of the catalog on the net.

A second contribution consists in the presentation of a system that implements these concepts. A guideline was to follow the standards as much as possible. An ActiveView application is compiled into a running application based on the following environment:

- We use the O_2 XML repository developed by ArdentSoftware [6] and its DOM interface for storing and querying XML data and methods.
- We intend to use the standard query language for XML when available. In the meantime (and in the examples of the present paper), we use a simple language inspired by Lorel [3].
- Each view is a multi-threaded Java program using the Java-DOM binding. We also use the notification mechanism provided by the O_2 system.[2]
- We intend to use for Web interfaces XML documents and XML browsers interacting with the views via Java remote method invocation. Until XML browsers offer the support we need, we use dynamic HTML with embedded Java applets. From a user viewpoint, an application presents a sequence of Web pages containing (modifiable) data and buttons, in a standard manner. The pages may evolve dynamically (e.g., new promotions may appear).
- A running application can be automatically generated from a view specification. We offer flexible means to customize such applications.

We already implemented a first prototype that was supporting only very partially the ActiveView features. The first prototype on top of O_2 was based on ODMG data and OQL. We were lead to XML mostly because a lot of data relevant for Web applications do not have the regular structure of ODMG and because of the (future) existence of many standard tools for XML such as sophisticated editors and browsers. In this paper, we describe the system that we are currently implementing. We mainly focus on the functionalities it provides.

The paper is organized as follows. Section 2 introduces active view applications. Using an example, it illustrates the needs for the various functionalities of our system, presents the data model and query language on which we rely and the architecture of a running application. In Section 3, we show how the data part of the application is specified before considering active features in Section 4. Section 5 discusses the default application generated by the system and different ways to customize it. A more detailed description of the user interface to *ActiveView* is beyond the scope of the present paper.

2 General framework

In this section, we introduce active views. We briefly give some minimum background on XML. Finally, we present the architecture of the system.

[2] This mechanism existed already for C++ and we had to adapt it to the Java-O_2 binding.

2.1 Active views

An ActiveView application allows different users to work interactively on the same data in order to perform a particular set of controlled activities. Before getting into details, let us illustrate the need to support such applications by considering an example. An electronic commerce application, say, a virtual store, typically involves several types of *actors*, e.g, customers and vendors. It also involves a significant amount of *data*, e.g. the products catalog (typically searched by customers) or the products promotion information (typically viewed by customers and updated by vendors). Observe that each of the actors may *view* different parts of the data (e.g. a customer can only see his/her own orders and the promotions relevant to his/her category, while vendors may view all the orders and promotions), each may perform different *actions* on the data, and have different *access rights* (e.g. promotions can be updated only by certain vendors). Also, the requirements for *freshness* of data may differ, e.g. when promotions are updated, we may want to immediately refresh the customers screen with the new data, but when a catalog item is updated we may not want to disturb a customer viewing the (old version of the) catalog; i.e., only when/if the customer actually issues an order for an updated item should he be informed of the change.

Each actor typically performs several *activities*. E.g. a customer may be *searching* the catalog, *ordering* products, *changing* a passed order. Observe that in each of these activities, we may expect to show a different Web page to the actor that possibly includes only part of the data and actions available to that given actor. Observe also that actions performed by an actor in a particular activity may initiate other actions. For instance, when a customer orders a product, we may want to update the stock. Finally, note that it may be interesting to log some of the actors operations, providing a *trace* for later analysis or to settle possible disputes.

As mentioned in the application, although we are primarily motivated by electronic commerce applications, the ActiveView system applies in general to a wide (and growing) range of Web applications. The main emphasis is on *declarative specification* and automatic generation of the application (by compilation) rather than producing large amounts of application specific code. We will first focus on the declarative specification. We will see in Section 5 various ways to customize the application that is automatically generated.

As mentioned in the introduction, an ActiveView specification is a declarative description of an application. It specifies, for each kind of actor participating in the application: (i) the available data and operations, (ii) the various activities, and (iii) some active rules. Thus, the general specification of an application has the following form:

ActiveView application application_name

ActiveView actor-kind$_1$ **in application** application_name
    view data specification
    methods definition
    activities specification
    active rules **...**
ActiveView actor-kind$_n$ **in application** application_name **...**

Such a specification is compiled by the ActiveView system into some actual application that allows the different users to perform the given set of controlled activities, working interactively on the specified data. An ActiveView application may of course use an existing application of the repository and in some ways can also be seen as a means to export to the Web an existing database application in a controlled manner.

We will detail in the following sections the syntax and semantics of the various parts of a view specification. In the remaining of this section, we briefly introduce the XML data model and query language on which the system relies and then give an overview of the architecture of an ActiveView application.

```
<catalog>
<name> the catalog </name>
<dept> <name> Books </name>
        <item myid="b1">  <name> Leagues under the sea, J. Verne </name>
                          <price>  4.75 </price>
                           ....
                          <suppliers supps="s1 s2" />
                          <seealso otheritems="b2 b3"> Books by the same author </seealso>
        </item>
        <item myid="b2"> <name> Around the world in 80 Days, J. Verne </name>
          ...
        </item>
....
</dept>
....
</catalogue>
```

Figure 1: The Catalog

2.2 Data Model and Query Language

XML [23] is emerging as the new standard for data exchange on the Web. Its simplicity, the features and tools that it supports or will soon support (such as dynamic features, query language, sophisticated editors, browsers, etc.) makes it particularly attractive to both end-users and programmers. The database industry has recognized the potential of this new format and many vendors are now extending their technology so as to propose XML repositories (e.g., ArdentSoftware [6], Poet [16], ODI [13]). Given that and the fact that our goal is to support Internet applications such as electronic commerce, we chose this emerging technology as the basis for our work. We next briefly introduce (i) XML, (ii) the Document Object Model (DOM is an API to develop XML applications) and (iii) the XML query language that will be used throughout this paper for defining active views. For lack of space, the presentation is rather brief. Full definition of XML, DOM, and the query language constructs can be found in [20, 19, 3, 2].

XML Most readers are probably familiar with HTML (Hypertext Markup Language), the language currently used to describe Web pages. An HTML document consists of text interspersed with tag fields such as `<I> ... </I>` to describe the layout of the page, the inclusion of pictures, hyperlinks, forms etc. An XML document also features tags. However, unlike HTML, these tags do not provide instructions on how the document is to be displayed[3] but rather information on the logical structure of the document. To understand this, consider Figure 1 which shows the XML document corresponding to the catalog of some electronic commerce application. The `<item> </item>` tags are used to delimit the information corresponding to one catalog item, each item consisting of a sequence of tagged fields such as `name`, `price`, etc. Note that items can be given an identifier (e.g., `myid=''b1''`) which can be used to reference them within the document (e.g., in element `seealso`) or in some other documents. As a matter of fact, in our example, each item references a list of supplier elements (see field `suppliers`) that are defined in some other documents of our repository.

An XML document can be typed. This is achieved by means of a Document Type Definition (DTD). Figure 2 shows a possible DTD for Document 1. Note that each tagged field corresponds to one element description in the DTD. Also note the definition of the three attributes for the `item`, `suppliers` and `seealso` elements, resp. These attributes are used to represent references among elements. More precisely, `ID` is used to give identifiers to elements and `IDREFS` to reference such identifiers.

Typing is not a mandatory feature in XML, i.e. one can have documents, or document parts, without an associated DTD. However, since most optimization techniques rely on typing, it is realistic to assume that large XML applications will come with appropriate DTDs. In the sequel, we will denote element type definitions using the `Elem` suffix. For instance, `catalogElem` will denote the type definition associated

[3]This may be specified in a *style-sheet*.

```
<!DOCTYPE catalogDTD [
<!ELEMENT catalog  (name, department+)>
<!ELEMENT name  (#PCDATA)>
<!ELEMENT department  (name, item+) >
<!ELEMENT item  (name, price, picture, description, seealso, suppliers)
<!ATTLIST item itemid ID #REQUIRED>
....
<!ELEMENT suppliers  EMPTY>
<!ATTLIST suppliers  supps IDREFS #REQUIRED>
<!ELEMENT seealso (#PCDATA)
<!ATTLIST seealso otheritems IDREFS #REQUIRED>
....
]>
```

Figure 2: The Catalog DTD

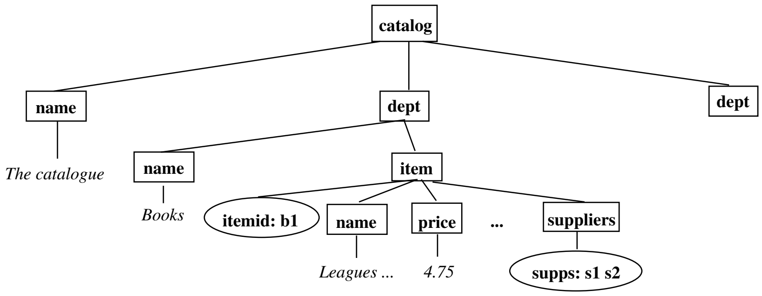

Figure 3: The Dom Representation of the Catalog

with the catalog element of the DTD given in Figure 2. Also, we assume that (using XML Namespace mechanism if needed) names of element types are unique in our context.

DOM DOM (Document Object Model) provides an API to develop applications using XML data. It gives a uniform way to view and access XML documents. It is a standard and for instance, ArdentSoftware and Poet repositories use DOM interfaces. In DOM, an XML repository is abstractly described as a graph, whose internal nodes represent data elements and whose leaves represent text or attributes. As expected, the parent-child relationship typically represents the component-of relationship. This is illustrated by Figure 3 which shows a partial DOM representation of our catalog. Rectangles and ovals represent, respectively, element and attribute nodes.

The DOM standard basically consists of a collection of classes and methods, providing generic access and update interface for the different kinds of nodes in the graph. For instance, the `getElementsByTag-Name` method, when applied on an element node, returns all the sub-elements (children) of the node having the given tag name. In the sequel, we assume that the interface of each element type can support a set of methods defined within the XML repository.[4] This feature is essential for most applications, e.g., to define a method on `Dept` element that will allow to update the price of all its items according to some change of VAT.

Query Language So far, XML does not provide a standard query language. However, there is a major standardization effort in that direction [8, 21, 22]. Our goal here is not to propose a new language or to compete against the up coming standard. Indeed, the ActiveView system will use this standard as soon as it becomes available. In the meantime, we rely on the Lorel language [3] to query DOM graphs.

[4]Note in particular that this feature will be supported by the coming release of the ArdentSoftware XML repository.

For example, the following query searches for `Item` elements whose price is less than 50 within the catalog document of Figure 3.

select	i
from	i **in** Catalog.$\star$.Item
where	i.Price $<$ 50

Note that we use the "$\star$" symbol to denote paths of arbitrary length.

The above query constructs a new DOM node whose children are the selected `Item` elements. But what data exactly, besides the nodes corresponding to the selected items, is considered a part of the user's view? Does it include all the DOM graph rooted at these nodes? And what about referenced nodes? (e.g. should the suppliers referenced by the selected items be included or not?)

As observed in [2], it is useful in a distributed environment to provide in the query language means for specifying the exact scope of a query result. Furthermore, as we shall see later, this will also turn to be useful for specifying appropriate access rights for the retrieved data. We follow here the syntax of [2] and add a *with* clause to queries. This extra clause describes, using path expressions, the subgraph reachable from the selected elements to be included in the view. For example, when added to the above query, the clause *with i.name, i.price* specifies that only the `name` and `price` elements of each selected item should be viewed. In general, a *with* clause may contain complex path expressions and introduce new variables. We will see some examples of that in the sequel. Observe that the *with* clause is a nonstandard syntax we are using. We believe that an XML query language will support such a feature, possibly as a separate clause as here or embedded in the *select* clause of the query.

2.3 Architecture of an application

The ActiveView system uses the O_2 XML repository that provides all the usual database features such as persistency, versioning, concurrency control, etc. It is based on a client/server architecture. An active view application (such as the one that will be specified in the next sections) consists of several independent clients communicating between them and with the repository server through notifications. Clients are programmed in Java, and communicate with the server using the DOM interface. Figure 4 shows the various components of one application (obviously, several such applications may run simultaneously on the same server). As can be noted, there are two kinds of clients: the *active view application manager*, which is unique per application, and the *active view* clients. These latter receive/send remote method invocations from/to the end-users *interfaces*.

The active view application manager consists of a set of modules managing: (i) connection and authentication, (ii) tracing, and (iii) active rules. More precisely:

1. The connection/authentication module is in charge of authenticating users and giving them the means to create[5] or quit a view (via the network).
2. The tracing module keeps a log of specified events. These events are generated by the application or by some views.
3. The active rule module manages a programmer-specified set of rules. (We will see their role in the application later on). These rules are fired according to events and may have impact on the repository and on some or all of the active views. They form the essential components to specify a business model.

[5] An active view is started from the Web using a particular URL.

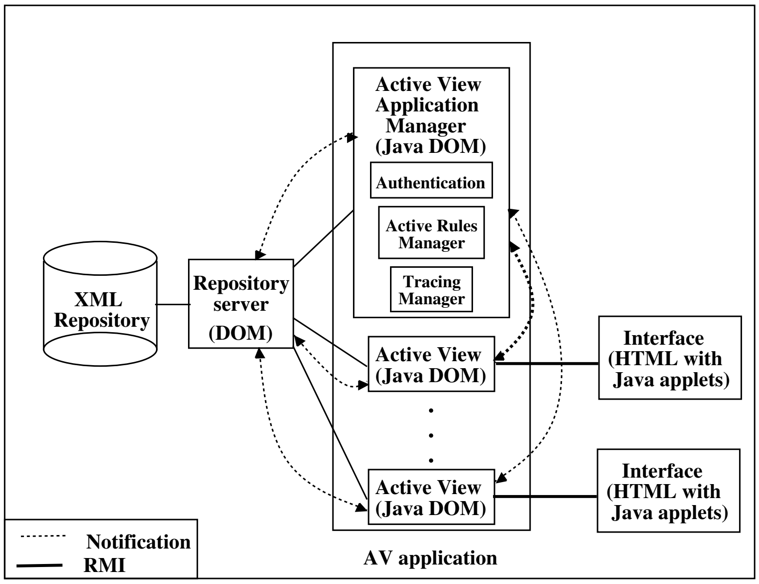

Figure 4: Architecture of an application

These last two modules rely heavily on a stream of notifications managed by the repository server that enables the interaction between views at run time. These notifications are generated according to the views specification. Two kinds of events can be notified: (i) events generated by the repository server after the creation/deletion/update of objects and (ii) user defined events generated by the clients. The O_2 notifications mechanism on which we rely has been partially developed by the Verso team at INRIA.

An active view is basically an object of our application. In the current version of the system, it is implemented in Java. The object belongs to a (subclass of a) particular class called *ActiveView*, which is an abstraction of the class we use in the actual implementation. This class contains certain instance variables (whose role will be explained later), including in particular the *owner* instance variable that is used for storing information on the user initiating the view. The class also has some methods such as transaction/commit/abort to handle a transaction mode, or init/quit/sleep/resume. An active view is generally related to an actual Web window opened by a user of the system. Some views independent of any interface may also be introduced, e.g., for bookkeeping. An active view has access to the repository as well as to some local data (the instance variables of the view object). It reacts to user commands and may be refreshed according to notifications sent by the server or the view manager. The methods available on a view depend on the users access rights and may allow him/her to read, load, write, etc. part or the whole the data it sees.

Users interfaces are currently being implemented as dynamic HTML documents with embedded Java applets. Our goal is to switch to XML as soon as XML browser supports the needed dynamic features. There is one HTML document per user and activity of that user. The applets are built on top of an API generated by the system according to the specification. Although the system generates default interfaces, the application programmer may redefine/customize them using the generated API that captures the semantics of the application.

In principle, the server, clients and interfaces may run on different machines. Typically, the interface is actually on a remote system. The view data is obtained by check-in/check-out, so repository changes are not in general immediately propagated to the view although they can be propagated, if specified by the programmer. On the other hand, the view and the interface see the same data.

3 Data and method specification

As stated in the previous section, an active view application involves several sorts of actors, each with a different view of the system. The specification of each kind of view consists of four parts that define respectively (i) data, (ii) methods, (iii) activities and (iv) active rules. In this section, we illustrate how the viewed data and the methods are specified and discuss related issues. The activities and active rules will be considered in the next section.

3.1 Data specification

An active view has local instance variables and derived ones defined using XML-queries. When specifying derived data, one also specifies the access modes (e.g., read). We illustrate this here using a very simple example in which we consider the interaction of three kinds of users: a set of *Customer*s and *Vendor*s and a single *Dispatcher*. A customer may browse the catalog, pass or modify an order. The dispatcher is in charge of assigning vendors to customers. At any time, there is only one active dispatcher. When someone tries to enter the system as a dispatcher, the person is simply turned down if a dispatcher is already in charge or if the person has not the proper qualification to be dispatcher. A vendor is mainly in charge of some customers and may interact with them, e.g., by offering them new promotions.

Consider first the customer view. A simple mode of importing data to the view is *read*. This is what is achieved for the catalog as follows:

let	catalog : CatalogElem
be	RepCatalog
with	catalog.⋆
mode	**read all**

This essentially imports the root element of the XML repository named *RepCatalog* (in our examples, the repository entry points are prefixed by *Rep*), and, as specified in the *with* statement, all the data in the *RepCatalog* document. That is, the entire DOM tree rooted at *RepCatalog* is imported. The *mode* clause specifies that we can read all we imported. This last statement is in fact not needed since the default on imported data is *read* for everything.

The possible modes beside *read* are *write*, *append*, and *remove*. An example of *append* in the *Vendor* active view is as follows:

let	promos: (PromoElem)*
be	RepPromos
with	promos.⋆
mode	**append** promos

This allows the vendors to see the set of promotions (implicit read) and to append new promotions to it. Removal of promotions from within the view is disallowed. The expression *(PromoElem)** indicates that the view document consists of a sequence of promotion elements[6]. To see a slightly more complex example, suppose vendors are also allowed to update the end-date of promotions. This is specified as follows:

let	promos: (PromoElem)*
be	RepPromos
with	promos.end-date X, promos.⋆
mode	**write** X, **append** promos

[6] In some cases it is possible to derive the type of an XML query [12]. We will ignore this issue here.

The query binds X to promotions *end-date*s. The expression *write X* indicates that these can be modified by vendors.

So far, we have defined only derived variables. Local instance variables are defined in the same way, except that they are not associated with a query specifying their value. Let us illustrate this with the variable *caddy* in the customer view.

local	caddy : (ItemElem)*
mode	**append**, **remove**

As illustrated above, the *with* clause is used to specify which data can be reached (i.e., viewed) from the objects bound to some query variables. Specifying this for each and every variable may be tedious, especially when the same element type is reachable from different variables and we want the same scope and access modes in all cases. One way to simplify the specification is to specify things at the element type level, i.e.. define for a given element type, the data that can be seen or modified when such elements are accessed.

To see an example, remember that the catalog contains references to suppliers. The following instruction, when added to the customer view, specifies that whenever a supplier element is included in the view, its name and full description are also included, in read mode (default), and its evaluations are included in an append mode (i.e., a customer may add his/her own evaluation of the supplier):

element	SupplierElem
with	self.name, self.description.$\star$, self.evaluations E
mode	**append** E

3.2 More on read and write

We consider next two issues related to read/write. The first has to do with the materialization of derived instance variables. The second is related to writes and transactions.

An issue is whether the views that we are using are materialized or not (loaded in the client interface). In general, we leave it to the system to decide whether a derived instance variable of the view is fully computed at initialization time, partially computed only (e.g., two levels of the tree of the query result are materialized), or computed (fully or partly) only when there are specific requests for data it contains.

The current default in the system is that instance variables are fully loaded at the initialization of the view. Also, when we read an instance variable, we read all elements specified by the *with* clause that are contained in the same document. Elements accessible by references to other documents are loaded only upon request. A similar philosophy is followed when reading an element based on an *element* specification. Observe however that certain applications may have some specific different requirements:

1. Consider an application that allows the user to check out a report to work on it at home, disconnected from the repository. Then the system should load the entire report. Suppose the report includes bibliographic citations that are references to some other bibliography document. The system should also load them immediately since the connection may not exist anymore when the user may request to see one of these citations.

2. On the other hand, consider a stock market application. We do not want to load in advance all trading rates since such information becomes rapidly stale. In this case it is better wait until a user explicitly requests a particular trading value before loading it.

To overrule the default, one can use two specific kinds of read modes, namely *deferred read* or *immediate read*. The first instructs the system to load elements only on demand, while the later indicates that elements should be loaded immediately when encountered. The keyword *read* can be replaced by one of these more specific modes anywhere in a view specification. For instance, one may add the keyword *deferred* to the *read* mode in the *Customer* specification of *catalog*. The elements contained in the catalog will then not be loaded at the initialization of the view but only upon explicit request from the customer.

Once some data is materialized and loaded into the client's interface, the user can view it or modify it, according to the specified access modes. Observe that these updates are not propagated to the database until explicitly requested by the user (i.e. by an explicit call to the *write* method that is part of the view interface). We will consider the problem of update propagation in more details later on. For now we only want to highlight the issue of transactions.

By default, a view is not in a transaction mode. Using base methods of the class *ActiveView*, a view can start a transaction and terminates it with an abort or commit. Reads are allowed outside transactions; so by default all reads requested by a view are *dirty*, i.e., no locks are installed. For updates, all updates *from a method call* in the repository issued by a view are required to be within a transaction. If an update is requested as a consequence of some method call and the view is not in transaction mode, an error is raised. The only exception is when the user issues an explicit call to the *write* method mentioned above. In this case, if the view is not already in a transaction, a new transaction is automatically started that lasts for the duration of the *write*.

3.3 Methods

The active view specification also includes definition of the methods available to the user in the given context. For example, assume that the dispatcher (more precisely, the user who is running a dispatcher view) is always aware of the connected customers in need of a vendor and of the active vendors. To support this, the dispatcher view may contain instance variables whose values are computed from the repository and describe the relevant customer and vendor sets. The dispatcher also has a method, namely *assign*, that allows to perform assignments, i.e., assign a customer to vendor. In the view specification, this method is defined as follows:

method assign(v: Vendor, c: Customer) **is** v→attend(c)

Note that the implementation of that method is specified in the view. But it essentially consists in calling some method known by the repository. Therefore, the view specification does not contain real code (besides XML queries) and is independent of any particular programming language. The methods in the XML repository may be in Java or C++ or in any language supported by DOM and the particular repository.

When activated by the application dispatcher, the above method will send a message to one specific vendor. This message will entail the execution of some code within the active view corresponding to the vendor (see Figure 4) and, potentially, the vendor interface will be modified. Another, sometimes more interesting way to modify a client interface, is to have it run its own code, independently from the repository server or clients. In order to do, the ActiveView system allows the declaration of *remote methods*. For instance, one can specify the following remote method in the Vendor view:

remote method new_customer (c: Customer)

and change the *assign* method to also invoke *new_customer*. Note that the code of the method is not specified. In the default application, it corresponds to a simple *message* with the name of the method as title and the parameters of the method as content. Thus in the default application, the vendor who is assigned to a new customer will receive a message titled "new_customer" with an object *Customer* in it. The visible portion of this object will have to be defined with an *element* statement in the *Vendor* specification. Now, as will be explained in Section 5, this default application is in fact built on top of an API, generated by the application compiler. A programmer desiring to customize the interface has the means to redefine the method *new_customer* so that its own code is executed when this call is received by the interface.

3.4 Access Rights

It should be stressed that access rights have to be much more sophisticated in the kind of Web applications we are targeting, e.g., electronic commerce applications, than in most standard repository applications. We therefore provide the means to attach an access predicate to any instance variable or method of a view. This predicate may use, for instance, the actual content of the view data and the user identification.

In general, access rights determine the modes of instance variables (e.g., read/write), and determine if methods are active or not. For instance, a customer may be disallowed to submit orders if its approved credit is negative or if he/she is in the group of blacklisted clients. This can be implemented by adding an access control clause to the specification of the method *submit_order*:

method	submit_order() **is** self.owner→pass_order(neworder)
if	(owner→approved_credit() >= 0 **and** !("blacklisted" **in** owner→group))

In the *if* clause, we allow arbitrary XML queries returning a boolean.

Remark : Access right may be quite expensive to check. In many cases, the access rights will depend only on the parameters of the initialization procedure of the view. The access rights may then be evaluated once and for all during the initialization of the view. This optimization may result in enormous gains in performance. But observe that it may be difficult to detect that it is indeed the case that some rights depend only on immutable values of the view. So, to indicate to the compiler that access writes have to be computed at initialization only, one can use the clause *static if* instead of *if* to specify the access rights. ◇

To conclude this section, we consider the issue of update propagation.

3.5 Update propagation

In one direction, when a derived attribute is modified in the view and a *write* is requested, we have to propagate the change from the view to the repository. In the other direction, when the repository changes and a *read* is requested for some derived attribute whose value was already previously computed for the view, we have to propagate the changes from the repository to the view. The detection of changes will be considered in Section 4.

Let us consider first the *view update* problem. We touch here upon one critical issue in databases. Most works on view updates have focused on updating views defined by complex relational queries involving joins and projections, e.g., [7]. This is a quite complex problem that we avoid here by an extensive use of objects and simply disallowing updates to views defined by too complex queries.

We maintain a correspondence between the repository and the modifiable portion of the view. In the best cases, an atomic value in the view (say a string) corresponds to an atomic value in the repository and the modification of the value in the view is easily propagated to the repository. In other cases, a view value does not have any exact correspondence in the repository (e.g., it is defined as a selection on some collection). We can still accept the update and propagate it to the repository in some simple unambiguous cases. In many cases, we simply disallow the update through the view unless the application programmer provides a method for it – and in that case, the system is not responsible for correctly propagating the update.

We mention next two important cases where the update is propagated, more on the subject can be found in [1]:

1. Strict correspondence between two collections: this is the case when each element in the repository collection has a corresponding element in the view, e.g., a set of objects and the same set of objects with a different interface specified by the view. Updates to elements are propagated when possible, i.e., when the propagation is defined at the element level. If an element is removed from the collection, we remove it from the corresponding repository collection. If one is inserted, we construct a corresponding element (eventually with a default value) if possible.

2. Partial correspondence between collections: this may happen if the view is obtained by filtering only some elements in a repository collection (and possibly restructuring them). This is a case quite frequent in practice that raises a number of issues. The main difficulty is upon insertion of an element (in the view) to verify that the view element that has been inserted when propagated to the repository actually results in an object that passes the filtering test. If this is not the case, the update is simply rejected.

To illustrate the previous discussion, we consider a definition of *Customer* where a customer may modify his orders by updating an instance variable *myorders* defined in the view:

let	myorders: (MyOrderElem)*	
be	**select**	O
	from	O **in** RepOrders
	where	O.buyer = owner
with	O.*	
mode	**write all except** O.buyer	

A customer may now update the result of a filtering of the entire set of orders, *RepOrders*. The system maintains a correspondence between the elements of *myorders* and those of *RepOrders*, so that an update to such elements can be propagated to the repository. The removal of such an element would result in removing the corresponding element from *RepOrders*. The addition of a new element would result in adding a new order to *RepOrder*. Observe that the customer cannot modify the identity of the buyer who issued a passed order because of *except O.buyer*. However, as it is defined here, the customer may in principle add a new order as if it was issued by another customer by simply putting the description of another customer in the *buyer* field. This could be anticipated using active rules to be defined further.

Let us now consider the *repository update propagation* problem. An issue is the re-computation of some view values when the database changes. Suppose that a user has loaded in a view the catalog and asks to re-read this catalog at some later time. The sequence of updates between the two reads is not available. One may consider using the repository versioning mechanism. It would suffice to compute the Δ between the version the user has and the current version. Clearly, sending a Δ instead of the entire value may result in large saving in communication. Versions are not considered in the ActiveView system for the moment. We will see further how, in some cases, we may have the list of updates and consider directly the incremental maintenance of the view.

4 Active features

The previous section considered the static part of the view definition. We now illustrate how a view can be made active. Note that we touch here a subject in close relation with workflow management. The main difference between our approach and workflows is the importance we give to data specification.

Workflow systems give declarative means for specifying the operations flow, but the data involved is typically described in a very abstract manner, often disconnected from the description of the flow itself. This makes the analysis of the connection between various pieces of information, their sources, and mutual effect of operations on them, very hard. A good example is the newly adopted standard, UML [14], which includes state-charts and activity diagrams for business process modeling but where data objects, whose value are used or determined by the action, are modeled only as parameters of some messages.

Most workflow models available today lack a semantic definition other than the operational definition implied by the tools [10]. The meta-model proposed by the Workflow Management Coalition [18] connects input and output data to activity, but doesn't provide implementation details. Due to lack of concrete guidelines, workflow management system as promoted by industry, uses a process-centric approach. Those models are extended by customized features for modeling and executing applications, but do not have adequate support to satisfy the modeling and correctness requirements of advanced applications [5]. Some of the deficiencies include lack of support to keep track of data dependencies for distributed workflow, lack of support to control concurrent accesses to objects managed by non-transactional activities, insufficient support for recovery etc.

In our system, activities are specified in two steps. First, for each kind of actors (i.e., each view), the programmer declares a set of activities along with the data and methods that can be used in those activities. Then, a set of rules specifies the semantics of the view. Typically, rules specify how to react to certain events. Two particular kinds of rules are of particular importance: (i) notification rules that allow to be notified that certain events took place, and (ii) *tracing rules* that allow to keep a log of some selected events.

We next consider the declaration of activities, the general rules, then the notification and tracing rules.

4.1 Declaring Activities

From an end-user viewpoint, each activity corresponds to a hypertext document with some data and buttons. For instance, the activity *search* defined within a customer view will show the catalog, some promotions, a collection of selected items (i.e., a caddy) and some buttons allowing the user to search the catalog, add some items to the caddy, order (i.e., change activity) or quit the application. This is specified as follows:

activity	search **includes**
	catalog, promotions, caddy
	search(), goto_order(), add_to_caddy(), quit()

A default stylesheet is attached to each activity. More generally, an activity declaration has the following form:

activity	<activity-name> **includes**
	<variable-name>* <method-name>* | **all**

where <variable-name> (resp. <method-name>) denote variables (resp. methods) specified within the view where the activity is being defined. The keyword *all* may be used to specify that all variables and methods of the view are visible.

It should be noted that although a given activity sees only a specific part of the view stated in its definition, *all* the ActiveView data is maintained (at least virtually) by the system. This allows different non consecutive activities to share data, and is in particular useful when a user resumes some activity after having gone temporarily to another one.

4.2 Rules

We consider here very standard active rules. Rules are specified inside a view. If global rules need to be considered in an application, one can clearly add a particular view that does it in the style of the *Dispatcher* of our example application. This provides some modular way of specifying active rules.

The specification of a view may therefore contain some active rules. The rules are processed by a *rule manager*. Rules are expressions of the form:

on <event> **if** <condition> **do** <action>

The components of an active rule are defined as follows:

- the events are (remote) methods calls (e.g., switch of activity), operations on instance variables or objects (i.e., write/read/append/remove) and detection of changes;
- the conditions are XML queries returning a boolean; and
- the actions are (remote) methods calls, operations on instance variables or objects, notifications or traces.

We illustrate active rules with some simple examples. The discussions on variable changes, notifications and traces are postponed to the following sections.

Suppose that when a new order is issued, we want to modify the stock of the store. This may be achieved by a method, say *update-stock*. The following rule in the *Dispatcher* view may be used:

on	submit_order(owner,neworder)
do	neworder→update-stock()

(An absent *if* clause is assumed to be always true.)

Next, let us consider remote method calls. For instance, suppose that a remote method *missive* has been defined in the *Customer* view and that we want to send a particular welcome-back message to good customers when then start their *search* activity. This can be achieved by defining the following rule:

on	goto (owner, activity)
if	activity = Customer::search **and** "good-customer" **in** owner→group
do	owner→missive("Welcome back. We appreciate your business.")

Observe that the bindings of *activity* and *owner* in the triggering event is used by the *if* clause.

4.3 Notifications and change monitoring

Notifications are based on remote method calls that can be sent to the interface of a view to notify that certain events (as specified in the previous section) have occurred. An important kind of events are (potential) changes of an instance variable. In the application default interface, the detection of a change for an instance variable (or an object) results in changing the background color for the display of the variable. The notification of other events results in a message being displayed to the user with a "notification" icon. These messages resemble the remote method calls we already discussed. Indeed, notification may be customized in the same manner as remote method calls.

To see an example, suppose vendors need to be notified of submissions of important orders by customers they are in charge of. This is achieved by the following rule in the *Vendor* view:

on	submit_order(owner,neworder)
if	neworder.amount > 10000 **and** owner **in** MyCustomers
do	**notify-me**

In this statement, the keyword *notify-me* specifies that the particular event must be notified to this view. In some sense, the view is issuing a *subscription* to certain events. Observe that only the views that explicitly subscribe are notified.

Now, let us consider derived instance variable monitoring. It is easy to detect that a repository object has changed. If an instance variable is defined by a complex query, the situation is more intricate. To see an example, consider the instance variable *promos* in the *Customer* view. In order to have the customer be notified of a change in its *promos*, the following rule must be included (*changed_promos* indicates that the variable *promos* has changed):

on changed_promos **do** **notify-me**

The variable *promos* may change if a new promotion that applies to the particular customer is appended or deleted. It may also change if one existing promotion is modified. The view therefore maintains the list of objects whose change may affect the derived data. (In this case, the collection object and each element in the collection.) When such a possible change has been detected, two cases occur:

1. The derived data cannot be maintained incrementally. In this case a notification is issued. Clearly, this may result in false "alarms".

2. The derived data can be maintained incrementally. An incremental evaluation of the changes is performed in the style of [2] to see whether actually changes occurred. No false alarm may occur.

Notification are just warnings. Data can be updated by the user by clicking on a *read* button or, automatically, by a customized user-interface (Section 5). Even when an incremental evaluation has been used and the new value is known by the system, it is sent to the client only when requested. It is possible to include in the view a rule to actually force changes to be sent to the client whenever detected. For instance, one could use the rule:

on changed_promos **do** promos→read()

In this particular case, the derived instance variable is simple enough to be maintainable incrementally. In case (2), such a statement may be costly since each detection of possible change will trigger the full re-computation of the instance variable and its shipping to the client.

Remark The need to "monitor" instance variables or to "refresh" them when changes occur is encountered in many applications. We therefore provide syntactic short-cuts to specify such features without having to explicitly write the corresponding rules. Instead of using *let <variable>* in the view specification, one may use *let monitored <variable>* or *let fresh <variable>*. This results in generating the appropriate rules to notify changes and eventually (in the case of *fresh*) trigger automatically a read when a change is detected.

4.4 Traces

Typically, database systems provide *logs* that are (i) low level and (ii) difficult or impossible to access. Yet, tracing the run of a business transaction is essential for electronic commerce applications. This may be required for legal reasons, to be able to handle eventual disputes between the participants, or to analyze buying patterns. In ActiveView, events and rules are at the core of the *tracer* module. We explain this next.

In a view specification, in the same manner we request to notify the view of certain events, we can request to *trace* them, i.e., notify the tracer. For instance, we may request to trace all order submissions:

on submit_order(owner,neworder) **do** **trace**

If such a statement is included, the tracer will be notified of the new orders and record them in the repository. The tracer also records the parameters and the time of the event. The log can then be viewed as a partial history of the activity of the application and can be queried. For instance, the following query returns the orders of a particular customer in 1998:

select	O
from	O in Trace.submit_order
where	O.neworder.buyer.name = "J. Doe" **and** O.date = 98

In this query, *Trace* is an entry-point to the XML repository that allows to access traces.

5 Default interface and customization

Compared to traditional database applications, electronic commerce and, more generally, Web applications are evolving very rapidly according to new commercial needs. For this reason, the ActiveView system not only supports the declarative definition of views and activities on the server side, but also a fast exploitation on the end-user side by generating *default user interfaces*. In this section, we briefly discuss the default interface, then various means of customizing the application and its interface.

5.1 Default User-Interface

When a user starts a new active view client by following a URL link, e.g.

http://www.activestore.com/customer,

a default HTML page is displayed and asks for identification information before proposing all possible activities that can be executed by the registered user. For example, all unregistered clients may be able to browse the catalog (activity *search*), but only registered users can also buy the selected products (activity *pay*).

Activities form the basic interface metaphors perceived by an end-users (e.g. client) interacting with the ActiveView system. Each activity is represented by a distinct HTML page which displays all accessible variables and methods in form of simple applets/buttons. (As previously mentioned, we intend to move soon to XML and stylesheets.) Applets are necessary to implement active features for calling methods and modifying/monitoring variables by communicating with the system via Java RMI calls (see architecture in Figure 4). Essentially, each variable corresponds to an applet editing the variable value and providing specific buttons for all access modes (read/write/append/remove) defined in the view.

The editing of view variable is an interesting issue. The ActiveView system is based on the XML document model and some XML query language for representing and querying data. This also means that variable values are XML fragments that should be *dynamically* merged into a comprehensive and uniform XML document. Whereas this issue is outside the scope of this paper, we believe that future XML browsers will propose some script language for modifying documents dynamically based on the DOM standard. Observe that a similar mechanism is already existing for HTML browsers in form of the JavaScript language.

View methods are called by simple button clicks. For example, in order to add a product to the caddy, the user calls a method *add_to_caddy* which adds a selected product to the caddy. Observe that this assumes to be able to select a product, e.g., in the catalog, and provide it as argument to *add_to_caddy*.

5.2 Application Customization

As defined so far an ActiveView application includes practically no code besides XML queries. Although we did not insist on that, it is clear that it may call repository methods that are implemented in conventional (DOM compatible) programming languages such as C++ or Java. Such code may be part of database application somewhat independent of the view application itself. View applications can also be customized in various ways at the cost of writing some *view specific* code. For instance, one may want to redefine the authentication procedure or the HTML page layout. (It is also obvious that several customization tasks, e.g. replace default authentication procedures, are available only to the system administrator(s).) Customization is briefly considered next.

Customizing view components Each Web client interface is communicating with an active view which is an instance of a subclass of class *ActiveView*. For instance, *Customer* is a particular subclass of *Active-View*. Instances of this class provide the necessary functionalities for logging into the system (init, owner, start_date), choosing among available activities and controlling the view status (quit, transaction, commit, abort, sleep, resume, timeout, see Section 3 for details). The class features also instance variables (i) the kind of the view (e.g. *vendor*, *client*), (ii) the owner, (iii) the date of creation, (iv) the current status (running, asleep), (v) the current activity, and (vi) the list of other available activities.

Customization essentially is possible by creating subclasses and overloading existing method codes. For example, the *init* method is executed when a new user logs into the system (creates a new view instance). It executes a private *authentication* method that verifies (by password or more sophisticated third-party authentication services) the identity of the user and fills in the value of *owner*. Both methods *init* and *authentication* are defined in the class *ActiveView* and may be redefined in a subclass, e.g., *Customer*, by the administrator, for example, to change the access right rule (Section 3), authorization mechanism or add additional preprocessing.

To see another example, consider *timeout*. Since active view applications run over the network there is no means to control the liveliness of a network client. Therefore, view objects come equipped with a simple timeout method that may force a view into *sleep* mode if the view has been inactive for too long. (The "too long" is specified by default.) The application programmer may decide to redefine this method in a particular subclass of *ActiveView* and indeed may also make it take into consideration some values of the view or specific resource parameters (transmit rate, client architecture, ...).

Customizing the interface One may customize the interface at several levels: presentation, method redefinition or total rewriting of the interface.

Users choose among various activities which correspond (by default) to different HTML and, in the future, XML pages. At the lowest level, it is obviously very easy to modify the presentation of an XML page by simply changing the stylesheet. Note that by doing so, one may hide certain functionalities of the particular activity.

Each interface is attached to a particular activity that specifies the available data and (remote) methods. Each such interface is implemented by an applet that communicates with a remote object that corresponds to that particular *Activity*. For instance, we may have an *Activity:Customer* class. It is possible to redefine the code of some methods of the class. As mentioned in Section 3, remote methods have to be implemented by the user interface. By default, remote methods simply display the parameters in the screen of the corresponding vendor. One may decide to modify this behavior. For instance, when a vendor receives the notification that a new customer is assigned to him/her, one may also want to automatically save in a local file the data about this particular customer.

Finally, one may want to completely redefine the interface to some activity, e.g., *Activity:Customer*. The API to the ActiveView system for this activity remains fixed. It is however possible to develop a Java applet (or application) that interacts with the ActiveView system via this particular API.

Acknowledgments We thank Victor Vianu, Brad Fordham and Yelena Yesha for works with one of the authors [4] that somewhat initiated the present work. The first prototype of the system was implemented by Sacha Arnoud, Mohamed Bani, Rhiad Dhaou and Frederic Hubert. Brendan Hills is thanked for his work on the new interface to ActiveView. We thank Guy Ferran, Sophie Gamerman, Jean-Claude Mamou (from ArdentSoftware) and Arnaud Sahuguet for discussions on this work. Finally, members of the Verso and Rodin groups at INRIA provided valuable comments, and in particular, Sihem Amer-Yahia, Jérôme Siméon and Anne-Marie Vercoustre.

References

[1] S. Abiteboul, S. Cluet, and T. Milo. A logical view of structured files. *The VLDB Journal*, 7(2), May 1998.

[2] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, and J. L. Wiener. Incremental maintenance for materialized views over semistructured data. In *Int. Conf. on very Large DataBases (VLDB)*, New-York, August 1998.

[3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1), April 1997.

[4] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *ACM PODS*, 1998.

[5] G. Alonso, D. Agrawal, A. El Abbadi, Mohan. U. Kamath, R. Guenthoer, and C. Mohan. Advanced transaction models in workflow contexts. In *Proc. Int. Conference on Data Engineering*, 1996.

[6] Ardent Software. *"http://www.ardentsoftware.fr"*.

[7] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

[8] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Xml-ql: A query language for xml. http://www.w3.org/TR/NOTE-xml-ql/.

[9] A. L. Furtado and M. A. Casanova. Updating relational views. In W. Kim, D.S. Reiner, and D.S. Batory, editors, *Query Processing in Database Systems*. Springer-Verlag, New York, 1985.

[10] M. U. Kamath and K. Ramamritham. Bridging the gap between transaction management and workflow management. In *In NSF Workshop on Workflow and Process Automation in Information Systems, Athens, Georgia*, 1996.

[11] A. M. Keller. Updates to relational databases through views involving joins. In Peter Scheuermann, editor, *Improving Database Usability and Responsiveness*. Academic Press, New York, 1982.

[12] T. Milo and D. Suciu. Type inference for queries on semistructured data. In *to appear in ACM PODS*, 1999.

[13] Objectstore. `http://www.odi.com`.

[14] OMG. Uml notation guide, version 1.1, 1 september 1997, 1997. www.omg.org/techprocess/meetings/schedule/Technology_Adoptions.htm.

[15] P. Picouet and V.Vianu. Semantics and expressiveness issues in active databases. *J. of Computer and System Sciences*, 1997. to appear.

[16] Poet. *"http://www.poet.com"*.

[17] C. Souza, S. Abiteboul, and C. Delobel. Virtual schemas and bases. In *Proc. EDBT, Cambridge*, 1994.

[18] WfMC standarts. The workflow reference model, version 1.1, wfmc-tc-1003,19-jan-95, 1995. http://www.aiim.org/wfmc/mainframe.htm.

[19] W3C. Document object model (dom). http://www.w3.org/DOM.

[20] W3C. Extensible markup language (xml) 1.0. http://www.w3.org/TR/REC-xml.

[21] W3C. Extensible stylesheet language (xsl). http://www.w3.org/Style/XSL/.

[22] W3C. The w3c query languages workshop, dec 1998, boston, massachussets. http://www.w3.org/TandS/QL/QL98/cfp.html.

[23] W3C. The word wide web consortium. http://www.w3.org/.

[24] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, San Francisco, California, 1995.

[25] Yelena Yesha and Nabil Adam. Electronic commerce: An overview. In Nabil Adam and Yelena Yesha, editors, *Electronic Commerce*. Lecture Notes in Computer Science, Springer-Verlag, 1996.