Declarative Specification of Electronic Commerce Applications*

Serge Abiteboul, Sophie Cluet, Laurent Mignet
I.N.R.I.A.-Rocquencourt, France
Email: <firstname>.<lastname>@inria.fr

Tova Milo
U. Tel Aviv, Israel
Email: milo@math.tau.ac.il

Abstract

"Combining the existing semantics of EDI within an XML framework also makes possible large-scale automation of electronic commerce." *WebWeek* `XML/EDI Home Page`

In this short paper, we consider the use of a declarative language *(the* ActiveView language*) for specifying electronic commerce applications and, in general, a wide range of distributed applications based on sharing data and exchanging messages. The fact that the language is declarative allows the fast design, development and deployment and the flexible maintenance of such applications. The prototype* ActiveView *system that we implemented acts as an application generator that produces customizable Web applications with persistent data and basic workflow management.*

1 Introduction

The development of Electronic Commerce applications has benefited from several standards: (i) The **W**orld **W**ide **W**eb as a support for data exchange; (ii) e**X**tended **M**arkup **L**anguage [7] as a universal means of describing data; (iii) **E**lectronic **D**ata **I**nterchange (E.D.I [5]) as a common dictionary for EC data. However, the design, development, deployment and maintenance of Electronic Commerce applications are still very costly, tedious and time-consuming tasks even if application generators exist. This is in contrast with an economic context that pushes for quick response time to market demands.

A similar situation in the seventies led the database community to a declarative query language to access data, namely SQL. We believe that, similarly for EC applications, the solution to many problems passes via the use of a declarative query language. Such a language is presented here. The modeling of the data is based on XML and a query language for XML. We use here [4] but plan to adopt the standard query language for XML when available. The ActiveView language focuses primarily on what we believe is critical in these applications: the specification of (i) modes of sharing data and (ii) of communications between the various actors.

To validate the motivating ideas and the language, we implemented the ActiveView system and tested some EC application. The prototype is based on a three-tier architecture :

- The first tier represents the repository that stores XML data. (XML provides the desired flexibility in terms of semistructured data modeling [3].)

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*The present work was supported by the French Ministry of Research within the framework of the R.N.R.T. GAEL project, a joint project between MatchVision, the L.R.I (Universit de Paris Sud) and INRIA.

- The second corresponds to a Java server application;
- The third consists of the end-user interface, based on standard Internet browsers.

A more detailed description of the ActiveView system can be found in [2]. The example used in this paper are based on a Web catalog application that we demonstrated at VLDB.

The paper is organized as follows. Section 2 introduces the ActiveView applications. Section 3 presents briefly the language. The last section is a conclusion.

2 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*. Also, the requirements for *freshness* of data may differ. E.g. when a new promotion is entered, we may want to immediately refresh the customer screen so that the new promotion becomes visible. On the other hand, we may decide not to propagate all catalog changes to the user so as not to overwhelm communications with too many messages. When/if an order is issued, the user has to be notified if the price of some data actually ordered has been modified.

Each actor in an EC application 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 available data and actions for that given actor. Observe also that actions performed by an actor 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.

Functionalities The definition of an ActiveView application consists of:

1. the specification of the persistent application data (some XML document such as the product catalog). This is often provided by already existing applications;
2. the specification of the application in the active view language, namely AVL. Its compilation generates the run-time of the application and some default user interfaces.
3. the customization of the default user-interfaces.

Furthermore, it is possible to modify dynamically the application by adding and removing *active rules* (see Section 3).

As mentioned above, 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 for the given actor, (ii) the various activities, and (iii) some active rules. Thus, the general specification of an application is as shown in Figure 1.

ActiveView application application_name

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

Figure 1: AVL general specification

Architecture The ActiveView system uses Axielle, the ArdentSoftware XML repository that provides standard database support. An ActiveView application consists of several independent clients of the repository communicating between them and with the repository through notifications. These clients are programmed in Java, and communicate with the server using the DOM interface [6]. Figure 2 shows the various components of a running application. As can be observed on the figure, the *active view application manager*, controls the activities of the (possibly many) *active view* clients. Active views receive/send remote method invocations from/to the end-users *interfaces* which runs on standard Web browsers.

In principle, the server, clients and interfaces may run on different machines. Typically, the interface are on remote systems. The view data is obtained from the repository by check-in/check-out. Repository changes are not in general immediately propagated to the active view. Mechanisms to allow immediate propagation if needed are provided. The active view and the interface see the same data.

The ActiveView application manager consists of the following modules:

(i) The update manager receives notifications of changes from the repository. It notifies the appropriate views of changes they are concerned with. This mechanism is based on a notification mechanism provided by Axielle.

(ii) The active rule module manages a set of rules specified by the application programmer. These rules are triggered by events generated by the application such as method firings or repository updates. The rule mechanism uses the JSDT Protocol developed by Sun Microsystems.

(iii) The tracing module keeps a log of specified events.

Together these modules provide the business intelligence of the application.

An active view is basically an object of our application. In the current version of the system, it is implemented in Java. 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 application 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.

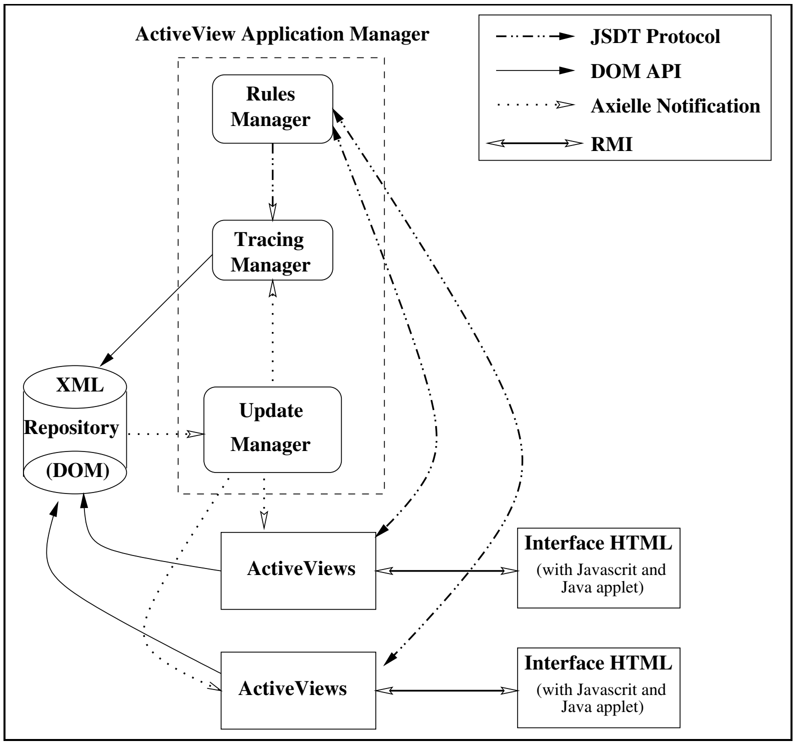

Figure 2: Architecture of an application

3 Activeview Language and User Interface

We describe briefly the language using a simplistic electronic commerce example in which we consider two kinds of users, namely, *Customers* and *Vendors*[1]. A vendor is mainly in charge of some customers and may interact with them, e.g., by offering them new promotions. More details on this application may be found in [2].

Data Specification A view may have access to persistent data (that exist independently of the particular actor) and transient data (that exist only during the actor's lifetime). The persistent data of a view are defined using XML-Queries [4] over the repository. Sophisticate access modes are provided. Consider, for instance, the *Customer's* and *Vendor's* views of the catalog defined as follows:

let	catalog : (CAT)⋆
be	{select $m.CATS.CAT
	from $m in $catalog.META_CATS.META
	where $m.NAME == 'Musical Instrument' }
with	**self**.⋆ X, X.ARTICLE Y, X.NAME Z
mode	**deferred read** X, **immediate read** Y Z

Customer's catalog definition

let	catalog : (CAT)⋆
be	{select $m.CATS.CAT
	from $m in $catalog.META_CATS.META
	where $m.NAME == 'Musical Instrument'}
with	**catalog**.⋆ X, X.ARTICLE Y, X.NAME Z,
	Y.PRICE P
mode	**read** X, **immediate read** Y Z, **write** P

Vendor's catalog definition

where *CAT* stands for category and *CATS* for categories. Customers and vendors see the same data except that a vendor may change the price of a product. The keyword *let* introduces some persistent data. (The keyword *local* for transient data is not used here.) Observe in the example how access modes are specified based on variables with bindings provided by a query.

[1] This example is based on the ActiveView Demonstration on VLDB

The *with* keyword permits to specify XML subtrees that are accessible in the view. The possible access modes are *write*, *append*, *remove*, *immediate read*, or *deferred read*. They allow to specify whether (some) updates are allowed and whether when an object is encountered, it should be loaded immediately (i.e., should we load the subtree rooted at this DOM node) or not.

The monitoring of persistent data changes is supported via two modes. The modes specify which actions should be taken when data changes are detected. If the keywords *let monitored* are used in place of simply *let*, the view is notified immediately of changes to that particular data. If *let fresh* are used, data changes are immediately propagated to the view.

Rules An actor specification may contain active rules. Rules can also be added and removed dynamically to modify the behavior of an application, e.g., to introduce a new kind of promotion. Rules [8] are very simple expressions of the form:

on (**local**)? <event> (**if** {<condition>})? **do** {<action>}

The events are (remote) methods calls (e.g., switch of activity), operations on instance variables or objects (i.e., write, read, append, remove) and change detections. The conditions are boolean XML queries. The actions are (remote) methods calls, operations on instance variables or objects, notifications or traces. A rule that is specified as *local* concerns this particular view and no event from or to other views.

As an example, we can introduce a new rule so that a vendor always sees what his/her customer is currently viewing:

on	display(element)
if	{sender.name == MyCustomer}
do	{self.display(element)}

Methods and Activities Due to space limitations, we will not describe here methods and activities specifications. In short, methods are defined in Java and are used for instance to do computations (AVL together with the XML query language should limit the need of Java code to the minimum). Activities are defined by simply listing the required data and methods (e.g., in the *browse* activity, we need the *catalog* variable and the *search* and *order* methods).

Users interfaces are currently based on dynamic HTML documents with embedded JavaScript and a Java applet to communicate with the active view. We plan to switch to XML as soon as XML browsers support the needed dynamic features. As for now, we generate XML "prescriptions", i.e., XML documents with application specific attributes and elements that describe the dynamic features of the document. These prescriptions are then compiled to dynamic HTML. There is one HTML document per user and per activity of that user. The applet is built on top of a generic API. In particular, the applet is application independent. The system generates default interfaces. The application programmer may redefine/customize either the XML prescription files or their corresponding HTML pages and JavaScripts.

4 Conclusion

The language AVL that is sketched here is supported by the ActiveView system [2]. There are limitations to the current architecture and prototype:

1. The combined use of JAVA RMI and Web Proxy limits the Web availability of ActiveView applications;
2. the use of one ActiveView Java client per user brings a lot of flexibility but would not scale to thousands of users;

3. the current ActiveView notification mechanism is rather complex. It uses both a database-centered notification mechanism from Axielle and JSDT. Although it supports nicely the toy application we developed, we believe it would not scale to thousands of notifications per seconds.

Acknowledgements A number of people helped implement the ActiveView prototype: Bernd Amann, Sbastien Ailleret, Frederic Hubert, Amlie Marian, Bruno Tessier, Brendan Hills. We thank Victor Vianu, Brad Fordham, Yelena Yesha for works with one the author [1] that initiated the present work. Finaly valuable comments were provided by Sihem Amer-Yahia, Anat Eyal, Jean-Claude Mamou, Jrme Simon and Anne-Marie Vercoustre.

References

[1] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 179–187, New York, USA, 1998.

[2] Serge Abiteboul, Bernd Amann, Sophie Cluet, Anat Eyal, Laurent Mignet, and Tova Milo. Active views for electronic commerce. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 138–149. Morgan Kaufmann, 1999.

[3] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publisher, October 1999.

[4] Vincent Aguilera. X-OQL. Technical report, INRIA, Verso Project, 2000. http://www-rocq.inria.fr/˜aguilera/xoql.html.

[5] Secretariat for federal edi. http://snad.ncsl.nist.gov/dartg/edi/.

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

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

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