Active Views for Electronic Commerce

Serge Abiteboul [*] Sophie Cluet [†] Tova Milo [‡] Victor Vianu [§]

March 16, 1998

COMMERCE
1com−merce \'kaˆm−(,)ers\ n
[MF, fr. L commercium, fr. com− + merc−, merx merchandise]
(1537)
1: social intercourse: interchange of ideas, opinions, or sentiments
2: the exchange or buying and selling of commodities on a large scale
involving transportation from place to place
3: SEXUAL INTERCOURSE
syn see BUSINESS

2com−merce \'kaˆm−(,)ers, ke−'mers\ vi com−merced; com−merc−ing archaic
(1596)
:COMMUNE

Definition given in XWebster

1 Introduction

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 [YA96]. Electronic commerce also requires database support, since it often involves handling large amounts of data (e.g. product catalogues, yellow pages, etc.) and must provide transactions, concurrency control, distribution and recovery. In this paper, we combine a number of established database technologies, namely object-oriented databases [Cat94], active databases [WC95, PV97], and views, e.g., [FC85, Kel82, SAD94], to provide the needed basis for electronic commerce.

[*] Serge Abiteboul, INRIA, Rocquencourt, Serge.Abiteboul@inria.fr
[†] Sophie Cluet, INRIA, Rocquencourt, Sophie.Cluet@inria.fr
[‡] Tova Milo, Computer Science Dept., Tel Aviv University, milo@math.tau.ac.il
[§] Victor Vianu, UCSD, vianu@cs.ucsd.edu

We argue that a database approach can provide the backbone for a wide range of electronic commerce applications. Beyond a supporting role, databases can provide high-level specification of the semantics of electronic commerce applications, in the form of declarative specifications of business models. The main components of the system we propose are as follows:

1. Object databases: the prime motivations for using an object database [Cat94] are (i) these systems are more open to the network via Java bindings and Corba bridges; (ii) these systems facilitate the design of graphic interfaces (graphic tools are generally object-oriented and objects are naturally suited to be displayed on the screen); and (iii) these systems facilitate the task of application programmers and thereby yield more programmers productivity. Since electronic commerce applications have intense network and interface components, and since they often require fast deployment and constant modifications, object technology is appropriate.

   We consider here the O_2 object database system that is compliant to the Object Database Management Group (ODMG) standard [Cat94].

2. Views: typically, users in these systems access data with different point of views depending on their access rights and on the nature of the activity they are presently involved in.

   We build here on top of O_2-Views [SAD94], a system we developed at INRIA. O_2-Views allows to specify views of an ODMG database that in particular allow to see values in the database that are computed instead of being stored (in the style of relational views), modify the object classification and object interfaces (via the concept of virtual class). Roughly speaking, we add to O_2-Views a sophisticated notion of access and update facilities.

3. Active databases: An active database is a database that can respond to certain *events* according to certain *triggers*. Very sophisticated active mechanisms have been proposed with modest success in industry. (Some database systems provide limited triggering mechanisms.) Electronic commerce applications are, by nature, very active, e.g., when a new order is received a number of actions may have to be started.

   We introduce active features in a somewhat minimal way here, using a notion of *monitored* objects which relies on the O_2 new notification mechanism. A prime motivation is the requirement to introduce the active mechanism without modifying the kernel of the database systems.

From a database perspective, the work presented here can be seen as an *application generator*. Indeed, we believe that active views enable a *declarative* specification of *certain kinds* of database applications. By declarative, we mean here that there is little (or no program) to write but that the description of the application is in a high level language (or a graphical interface). By *certain kinds*, we mean a wide range of simple applications involving (i) sharing of data and (ii) some cooperative work by a number of actors connected via the network. We present a number of target applications in Section 7.

Another important aspect of electronic commerce is the need to control tightly the activity of the users. This leads to using a sophisticated access mechanism. It also motivates a tighter management of some objects that we call *monitored objects*. In particular, we provide the means of logging calls to such objects. This is needed in particular in case of disputes between the participants in the business transaction.

Finally, we briefly describe how the Spocus framework, a very controlled model of business transaction introduced in [AVFY98] can be combined here. The idea is to limit dramatically the specification of business transactions so that a number of tasks of *program verification nature*, in general undecidable, become feasible. These are tasks such as business model customization (to allow the users to adapt the model to their own needs) or verification of temporal properties (such as *a product cannot be delivered if not previously paid for*). The Spocus approach also presents the advantage to be declarative thereby facilitating the task of explaining the semantics of the application to users. The combination with the very flexible and powerful environment of an object database allows the *atomic* tasks of the business transaction to be arbitrarily complex (e.g., involve complex payment mechanism or producing complex EDI forms).

The system (called *ActiveViews*) presented here is being implemented. In this paper, we mainly focus on the functionalities it will provide. The next section sketches the architecture of the system. Section 3 presents the definition language for active views and an example. Section 4 explains in more details the semantics of an application. In Section 5, we show how one can keep trace of the application activity. Section 6 introduces active rules and business models. A description of the interface to *ActiveViews* is beyond the scope of the present paper.

2 Architecture

An active view application is a program that allows different users to work interactively on the same data in order to perform a particular set of activities. At a given time, many active view applications may run on a particular database. For instance, one such application may manage business orders whereas another one manages the hot-line (bug reports, etc.). Although the various applications may share some data, each application is likely to be using mostly the proper data it manages.

To create an active view application, the programmer specifies different types of views (e.g., for customers or vendors in a business process) and the way they interact. The specification is then given to the application generator that produces the code in charge of creating/deleting/managing multiple instances of active views. This code consists of a main module called *Active View Manager* and, at run time, of several active views. This is sketched on Figure 1.

We will illustrate our presentation of *ActiveViews* with the specification of a simple electronic commerce application. At some point, the application is running and a number of vendors and customers are registered active views. These are human beings that are interacting, via active views, with the system and among them. New customers can connect via the Web and enter the game after some authentication phase. Another human being (the dispatcher) observes the arrivals and departures of customers and vendors, and assigns vendors to the customers. The system allows to specify in a simple and declarative manner such interactions. Most importantly, some active rules allow to *react* to some situations. One can for instance replace the human dispatcher or vendors by some rules[1]. But first let us explain the various components of the system.

The active view 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[2] or quit a view (via the network). Users can also put their views to *sleep* or reactivate them.
2. The tracing module keeps a log of specified events. These events are notified by the server according to the application needs.
3. The active rule module manages a programmer-given set of rules. These rules are fired according to events and may have impact on the database and on some or all of the active views. They form the essential components to specify a business model.

These last two modules rely heavily on a stream of notifications from the server to allow the interaction between views at run time.

An active view is an object of some subclass of the class *ActiveView*. A view persists in the database from the time it is created until the user quits. It 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.) The communication between the Web window and the view is managed by a cgi-script, using the view logical identifier. An active view has access to the database 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, lock, etc. part or the whole the data it sees.

The system is implemented as a standard application of the O_2 system. It uses most of the novel features of O_2 Version 5.0 and, in particular, the notification mechanism. The view manager and each active view corresponds to one O_2 client. Clients interoperate via the notification mechanism. The communication with interfaces is done via the network.

[1] One could also save the effort of building the interface by having only virtual actors involved `:-)`.

[2] An active view is started from the Web by asking for a particular URL. The system provides a form asking for the name of the database system, the name of the view application, and the type of the view.

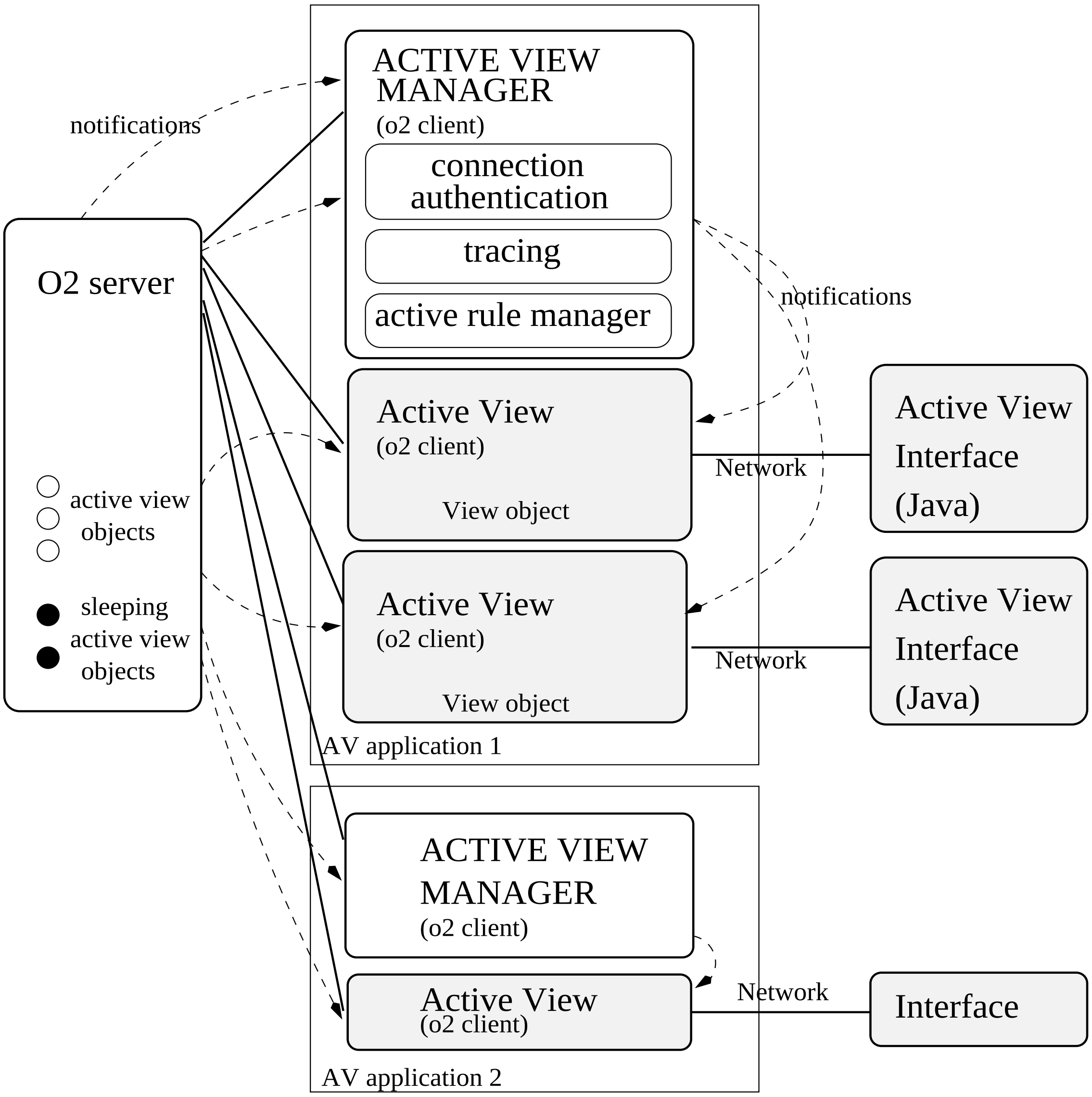

Figure 1: Architecture

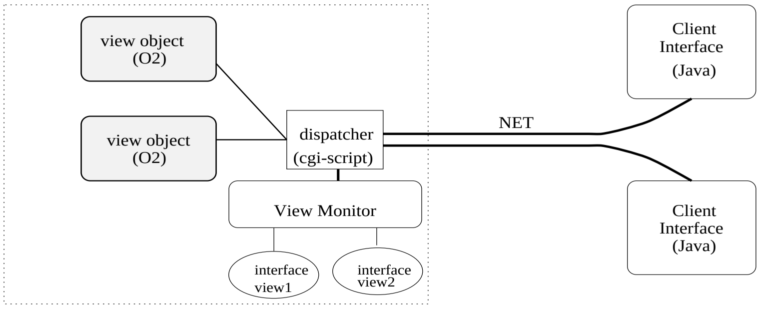

Figure 2: Interface architecture

In principle, the server, clients and interfaces may run on different machines. The view data is obtained by check-in/check-out, so database 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.

The system comes equipped with a default interface that consists in a Java application that interacts with the database via a cgi-script. This main mode of interaction that we are considering within our active view system is presented in Figure 2. Intuitively, the interface displays the instance variables of the views, eventually in editable mode, and buttons for the methods supported by the view.

ODMG and notification The implementation of the *ActiveViews* system is on top of an ODMG object database system. We are using the standard functionalities of these systems with a notable exception, namely *notification*, that is provided by the O_2 system. An implementation without the support of a notification mechanism would be much more intricate. However, (1) we believe that this is an important feature that should be made part of the ODMG standard, and (2) it is possible to simulate a notification mechanism but that would, of course, be less efficient than one built directly in the kernel of the engine as in O_2.

Remark 2.1 (Alternative architectures) We are also considering the implementation of the view as a Java applet. More precisely, not only the interface, but the active view itself would be in Java. This would present the following advantages: (i) the view can run on any machine supporting Java, and (ii) with some extra limitations, one could also consider views running on some minimal processors such as Java cards, see, e.g., [Gem], or cellular phones.

In the opposite direction, we are also investigating a very lightweight interface. In this last interface, the user interacts with the system via XML documents [W3C]. The documents are generated and interpreted based on the YATL [CDSS98] translation framework. YATL proved to be successful for customizing the display of ODMG data into data exchange formats. The XML approach seems particularly promising since (i) it allows defining active documents, and (ii) it would permit having several presentations of the same view by simply changing the style-sheet. □

3 Specifying an application

In this section, we describe the main component of the specification of an (active view) application. In the remaining of this paper, when we talk about application, we always mean active view application.

The system comes equipped with a class called *ActiveView* which provides the minimal set of features required by an active view application. To define an application, the programmer mainly refines this class in several subclasses, one per actor in the particular scenario. The definition of an application is illustrated by an example (that we will discuss in some details) in Figure 3. First, we describe the class *ActiveView*.

3.1 The class ActiveView

Its interface is as follows:

```
class ActiveView
   owner : Person;
   start-date : Date;
   status : Status;

   private method authentication();
   public  method init();
           write();  refresh();
           transaction(); commit(); abort();
           read(); lock(); time-out();
           quit(); sleep(); sleep-lock(); resume();
```

The class features three instance variables (i) the owner of a view object, (ii) the date of creation of the object and (iii) its current status. These allow to record the following information:

The owner of a view object is a *Person* that may belong to one or several groups. Group membership and person identity are used to enable the implementation of different access rights policies for view objects (see Section 4.5). This attribute is filled in by the *init* method.

Start_date simply indicates the date of creation of an *ActiveView* object. It is also filled in at creation time. We will illustrate its use in the sequel.

Status is a predefined class that allows to record various informations about the status of a view as a whole, but also, as we see next, of its various instance variables.

A user may refine class *ActiveView* by adding instance variables with different behaviors. For instance, the view used by a customer in a business application may have two instance variables representing (i) a particular order the customer is currently working on and (ii) the list of his/her pending orders. We may want the object corresponding to the first to be locked. On the other hand, it may be preferable not to lock the list of passed orders since someone else may, for instance, decide to modify one of the pending order (e.g., the delivery service when the product is delivered) or remove an element of the list (e.g., the financial services when the product has been paid for).

The variable *Status* records the status of the instance variables in the view in the following fields:

```
locked: set(AttId);      /* which instance variables are locked? */
stale : set(AttId);      /* which instance variables are stale?  */
write-wait : set(AttId); /* which instance variables have been   */
                         /* modified in the view but not yet     */
                         /* in the DB                            */
```

Other fields are used to record the global status of the view:

```
intransaction : TransId; /* if the view is in a transaction      */
                         /* it holds the transaction identifier  */
                         /* else it is nil                       */
asleep : Bool;           /* true if view is asleep - see below   */
```

The status of a view changes either when the user invokes some methods (e.g., *write* for *write-wait*, *sleep/resume* for *asleep*, *transaction/lock/commit/abort* for *intransaction*, etc.), or when the view is notified of some events by the database server (e.g., for *stale*). Consider the issue of detecting changes that affect the instance variables. If the instance variable contains a single object corresponding to a database object, it suffices to be able to detect updates to the database object, i.e., release of a write lock on the object in the database at commit of some transaction. On the other hand, if the content of the instance variable is the result of a query, we need a more sophisticated mechanism. For variables defined by complex queries, we will not attempt to maintain the *stale* field. In such cases, the system will simply disallow the specification of complex modes based on stale-ness, see *read fresh* further on.

The methods of Class *ActiveView* are explained next:

Authentication, init: When a view is created, the init method is called and it calls the *authentication* method that verifies (by password or more sophisticated methods) the identity of the user and fills in the value of *owner*. Both methods *init* and *authentication* can of course be redefined in subclasses if necessary. Note that being private, method *authentication* can only be called by a method of Class *ActiveView* or one of its subclasses.

Write, refresh: When *write* is activated on a view object, it forces a write of all its instance variables to the database and similarly for *refresh* that forces a read. As we will see in the next section, a view may have several *write/refresh* methods allowing to act on part of or on the whole view.

Transaction, commit, abort: We mention briefly how transactions are used in this framework. We rely primarily here on the transaction mode of the database system. The view object has a particular method *transaction*. When activated, this method starts a new transaction. The transaction ends when a *commit* or *abort* method is activated.

All updates on the database issued by a view are required to be within a transaction. If one update is requested and the view is not in transaction mode, this is in general an error. The only exception is the *write* methods that start a new transaction that lasts for the duration of the *write* only.

Read, lock, timeout: In many such applications, *dirty* reads (i.e., reads without locking the data) are used. So reads in active views are by default dirty unless within a transaction or if the user accesses an object contained in an attribute that has been defined to be locked when read (e.g., the order object a customer is updating). In this second case, a transaction is started. Since these applications run over the network there is no means to control the liveliness of the network client. Therefore, view objects come equipped with a *timeout* method. The application programmer may decide to redefine it in a particular view class and indeed may also make it depending of parameters of the view object.

Again, as for *refresh* and *write*, there may be several implementations of the *read*, *lock* and *timeout* methods to fit the needs of specific instance variables.

Quit, sleep, sleep-lock, resume These four methods allow to quit permanently or temporarily the view. The first three induce the termination of the transaction, if one is running. The last one may start a new transaction if needed. A view left with *sleep* and *sleep-lock* can be restarted through a call to the *resume* method. The only difference between *sleep* and *sleep-lock* is that objects that are read-locked remain so in the case of *sleep-lock*. (There is no way to interrupt the view and keep some *write* lock.)

3.2 Programming an application

An application is defined by a set of variables, a set of virtual classes (used to redefine the interface of an existing database class) and a set of *ActiveViews* classes.

In an application, there may be various kinds of active views. Each such kind corresponds to a particular subclass of class *ActiveView*, so an active view object has at least the interface of the class *ActiveView*. It particular, beside the instance variables, *owner*, *status*, *start_date*, it may have other instance variables that are either local (stored values) or derived from data in the database. The definition of derived instance variables is essentially in the spirit of (but much simpler than) O_2-Views. As in O_2-Views, we can define in a view

```
active view application my_business

    dispatcher: Dispatcher;
    customers: set(Customer);          /* Some Variables */
    vendors: set(Vendor);
```

```
    virtual class NewOrder from Order          /* Redefining a database class  */
       read all but number;          /* import most attributes */
       write product, quantity;          /* some are writable */
       init(owner: Person) {self.creator=owner;}          /* redefined method */
       available();          /* imported method */
```

```
    class Dispatcher inherits ActiveView          /* First ActiveView class */
             /* Instance variables */
       read fresh unattended_customers: list(Customer)
          has value select c from c in customers where c.vendor = nil order by c.start_date;
       read fresh available_vendors: set(Vendor)
          has value select v from v in vendors where v.status.asleep = false
             /* Redefined or new methods */
       init(); quit(); assign(Vendor, Customer);
```

```
    class Customer inherits ActiveView          /* Second ActiveView class */
             /* Instance variables */
       write neworder: NewOrder;
       lock write changeorder: NewOrder;
       write vendor: Vendor;
       read myorders: list(NewOrder)
          has value
             select (NewOrder) o from o in DBorders where o.creator = owner order by o.number
       read fresh promos: list(Promotion)
          has value select m from m in DBpromos where m->concerns(owner) order by m.number
       read fresh catalogue: Catalogue
          has value DBcatalogue;
             /* Redefined or new Methods */
       init(); quit(); tutored_by_vendor(Vendor); change_order(NewOrder); modify_order(); submit_order();
```

```
    class Vendor inherits ActiveView          /* Third ActiveView class */
             /* Instance variables */
       read fresh promos: list(Promotion) has value DBpromos;
       write customers: list(Customer);
             /* Redefined or new methods */
       init(); quit(); sleep(); in_charge_of_customer(Customer); entitle_to_promo(Promotion, Customer);
```

Figure 3: Active View Application: My_Business

some instance variables whose values are computed when needed via a *has value* clause. Also, we can import some objects from the database and specify a different interface in the view, via the notion of *virtual class*. Below is the grammar allowing to define instance variables. The how and why will be explained in the sequel. It suffices to know for the moment that the modes describe certain policies for controlling data changes. The control predicate is used for specifying access control. The *has-value* from O_2-Views is used when the attribute is computed.

```
<attribute-declaration> :- (<control>)* (<attribute>)*
<control>               :- (<mode>)* <control-predicate>
<mode>                  :- write | write immediate
                         | load  | load  immediate
                         | read  | read fresh
                         | lock
<attribute>             :- <attribute-name> : <type> has value <query>
                         | <attribute-name> : <type>
```

Let us now see how an application runs using the application specification given in Figure 3. There are mainly two non-exclusive ways of interaction between active views: using (i) updates and refresh or (ii) active rules. The second mode will be explained in Section 6. We now consider the first.

The example is very simple but is a good illustration of the way one can write a simple active application. It allows the interaction of three kinds of users: *Dispatcher*, *Customer* and *Vendor*. Their role and the way they interact is as follows:

A dispatcher is in charge of assigning vendors to customers. At any time, there is only one active dispatcher in the application and it is recorded in a variable of the application, namely *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.) To assign vendors to customers, the dispatcher (more precisely, the user who is running a dispatcher view) must be aware at all times of the customers in need of a vendor and of the active vendors. This information is given by the dispatcher view attributes *unattended_customers* and *active_vendors*. Their values are computed from the application variables *customers* and *vendors* and kept fresh (*read fresh*): i.e., any update on one of the application variables entails an immediate recomputation of the appropriate dispatcher attributes. The way this is done is explained in the next section.

The dispatcher decides which vendor should be assigned to an unattended customer by, e.g., browsing customers and vendors. The dispatcher disposes of a method, *assign*, that allows to perform assignments:

```
method assign(v: Vendor, c: Customer) in class Dispatcher
   {v->in_charge_of_customer(c); c->attended_by_vendor(v);}
```

The dispatcher activates this method by simply clicking on the appropriate button and filling (dragging the objects to) the two parameters. Then the method updates both the vendor and the customer states through appropriate method calls. As will be illustrated, this will result in a refresh of the dispatcher view. (In particular, the particular customer will disappear from the list of unattended customers.)

A customer may pass a new order (*neworder*) or modify an old one (*changeorder*). When a user requests entering the system as a customer, the following method code, for instance, is executed:

```
method body init() in class Customer
   {owner = authenticate();
    if (owner->approved_credit() <= 0) quit();
    customers+=set(self);           /* update list of customers */
    neworder = new NewOrder(owner); /* prepare new order        */
    changeorder = nil;              /* in case of change order  */
    vendor = nil; }                 /* vendor in charge         */
```

Note that the method updates the application variable *customers*. As was explained, this entails a refreshing of the *dispatcher* view which, at some point, will eventually update the *vendor* attribute of the customer using method *tutored_by_vendor*, which, in turn, will entail a refresh of the dispatcher view.

Also, note that an object of class NewOrder is created. This object is local to the view and can be freely modified by the customer (*write*). When the new order is ready, it can be submitted to the database using method *submit_order*:

```
method submit_order in class Customer
   {if neworder->available()
       { transaction();
         DBorders+= list((Order) neworder);
         ((Order) neworder)->assign_number();
         commit();
         neworder = new NewOrder(owner);
         self->refresh(); }}
```

The method updates the database object *DBorders*[3]. This is done within a transaction and requires casting from the virtual class to the corresponding database class. Note that the view here directly activates database updates. This is the simplest way to guarantee consistency. We will see in Section 4.3 an alternative based on using view updates and propagations to the database.

At all times, a customer is aware of his/her list of past orders. This list may be stale. To refresh it, the customer must explicitly click on the button corresponding to the

[3]For reasons of clarity, all database object names are prefixed with the string "DB". This is by no means a mandatory feature

refresh method. As opposed, the catalogue and the list of promotions are maintained fresh (*read fresh*). As we will see, the vendor acts on the latter by adding the customer to the list of people entitled to some database promotions.

To modify a previous order, the user must first selects the appropriate order in the list *myorders* and updates the *changeorder* attribute using the appropriate method. Once this is done, a transaction is opened, the selected object is locked and the customer can edit it (*lock write*). When the order is ready, it can be submitted to the database using Method *modify_order*:

```
method body modify_order in class Customer
   {if changeorder->available()
            self->write-order((Order) changeorder);
     changeorder=nil;
     commit(); }
```

Observe that, with this code, the transaction is terminated after *modify_order*. The code of *write-order* omitted here simply forces a write of the modified order using a call to method *write* in the class *Order*. We do it here within a method in the class *Customer* to be able to trace such modifications. To modify an order, the customer simply has to drag an order object from the list of orders to the *changeorder* field. Note that this automatically opens a new transaction since this field has been defined in (read) *lock* mode.

Finally, the customer quits the system by calling the method *quit*:

```
method body quit() in class Customer
   {Promotion p;
    transaction();
    customers-=set(self);
    for (p in promos) p->remove_right(owner);
    commit();}
```

Observe the update to the application variable *customers* (a customer is leaving the system) and the call to *remove_right* that is used to remove this customer from the lists of potential promotion recipients.

A vendor is mainly in charge of sending appropriate promotions to the customers he/she is in charge of. At this stage, the definition of this class and how a vendor view is updated by the dispatcher (attribute *customers*) and the database (attribute *promos*) should be clear. Let us just illustrate how a vendor sends a new promotion to a customer (after having decided, by browsing, which promotion was appropriate).

```
method entitle_to_promo(p: Promotion, c: Customer) in class Vendor
   {transaction();
    p->add_right(c);
    commit();}
```

Note that the set of database promotions is updated. Due to the various *fresh* modes, this entails a refresh of both vendor and customer views.

As was illustrated, an application can be active without having to define active rules: the interaction between the different active views can be easily implemented through automatic refresh and database or local variable updates. We will see in Section 6 a more centralized way to define the activity of an application. But before, we explain in more details the various features presented here and introduce the attributes modes that have not yet been explained.

4 Semantics of an application

The active part of an application relies on the fact that all local objects as well as some database objects are monitored. We first explain how this is done. Then, we discuss (ii) read and fresh modes, (iii) write mode and update propagation, (iv) load policies and (v) access rights.

4.1 Monitoring Objects

Monitoring objects is an essential feature of the system. It is essential for (i) keeping (portions of the) views fresh, (ii) maintaining the application log and (iii) implementing active rules. The previous section illustrated (i), Sections 5 and 6 will explain respectively (ii) and (iii).

To monitor objects, we rely on the new notification mechanism of O_2. Without entering into details, an O_2 client may require to be notified of certain events that are either basic functions of the data store (creation, update and deletion of objects[4]) or explicit events initiated by some other clients. A typical use of notification is to provide some form of communication between different clients. By default, basic events notification is provided for all objects that are part of the application activity. We will see in Section 6 how the programmer requires to be notified of more advanced events.

The notification mechanism is essential for implementing the refresh. The O_2 server detects that a registered object has been updated and notifies the view manager. The view manager then decides which active views have been affected and notify them. In some simple cases such as an object being directly imported to an active view in *fresh read* mode, the active view can receive directly the notification from the server. Note that when an instance variable is modified by an event, but is not in fresh mode, it is added to the view list of stale attributes in *status.stale*. Upon receiving a notification, the view manager also finds out which events must be logged and which active rules must be fired.

[4] A limitation imposed by O_2 is that objects have to register to the notification mechanism. This may cause troubles for objects that are already part of the database and that have not been declared as notifiable. One solution is then to use surrogate objects defined in a virtual class.

4.2 Read and Fresh

Given the way objects are monitored, there is not much to be said about read and fresh mode. When in read only, an object is computed once at the creation of the active view. When the *refresh* method is activated, the view client re-computes its stale attributes. When in *read fresh* mode, the re-computation is performed at notification.

It should be observed that a change of some object in *read fresh* mode in some view may have dramatic impact on the entire view. In particular, some computed values may be affected, the mode of some instance variables may also change. This will be illustrated in Section 4.5.

4.3 Write and Update Propagation

We touch here upon one critical issue in databases, propagation of view updates (i.e., using writes instead of methods to update the database). Most works on view updates have focused on updating views defined by complex relational queries involving joins and projections, e.g., [BS81]. 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 database and the updatable portion of the view. In the best cases, an atomic value in the view (say a string) will correspond to an atomic value in the database and the modification of the value in the view will be easily propagated to the database. In other cases, a view value does not have any correspondence in the database (e.g., it is the join of two relations). We can still accept the update and propagate it to the databases in some simple unambiguous cases. But there is no religion to accept all updates to the view. 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 can be propagated, more on the subject can be found in [ACM93]:

1. Strict correspondence between two collections[5]: this is the case when each element in the database collection has a corresponding element in the view, e.g., a set of object 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 database 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 is a database collection (and possibly restructuring

[5]We mentioned read and write modes for instance variables. For collections, one also consider more modes that allow only limited updates to collections such as *append-only, update-elements*.

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 database 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 an alternative definition of *Customer* where the customer passes and modifies orders by updating an attribute *myorders*. This leads to (i) removing attributes *neworder* and *changeorder*, (ii) changing the mode of *myorders* which becomes *read write* and (iii) adding/removing some methods. Among the new methods we find *set-submit* that propagates the view changes to the database:

```
method body set-submit in class Customer
  { transaction();
    for (each o in myorders)
       if (!((Order) o) in DBorders)
           ((Order) o)->assign-number();
    self->write-myorders();
    commit();
   }
```

This method first assigns an order number to the newly created objects (that are not yet part of the database). Method *write-myorders* is a system supplied method generated by the specification of *myorders* in *write* mode. A call to *write-myorders* forces the updates of *myorders* to the database, i.e., it modifies the set *DBorders*. The transaction is then committed.

As usual, concurrency between several users lead to subtleties. For instance, suppose that myorders is now in *write, read fresh* mode. Suppose that someone tries to update this data. The system issues a warning that the data that I am editing has been modified by someone else and offers to (i) either load the fresher value, (ii) force my update, or (iii) attempt to merge the two updates. The merging of the updates is handled in an ad-hoc manner by the system and no particular semantics is guaranteed. Note that this happens immediately after someone updated the shared data because my view was in *read fresh*, and thus was notified of the update. In standard *read* mode, this would have gone undetected until I would have tried to write, thereby raising a conflict with the "adversary's" write.

Finally, let us briefly consider the mode *write immediate*. When using this mode, the user requires that any change on the data must be propagated to the database immediately. This is implemented using a method of class *ActiveView* that we have not yet introduced: *write-time*. This method specifies the periodicity of the writes and can be redefined. Writes are performed only on modified objects. For instance, using *write immediate* we could specify:

```
emergency : Alarm write immediate;
warning : Alarm write;
```

The fact that *immediate* is used means that whenever the value of *emergency* is changed in the view (and after *write-time* has elapsed), the corresponding database value is updated. In the usual *write* mode, the database value will be updated when the user explicitly calls the *write* method on the object.

4.4 Loading objects

As a first approximation, one can consider that the views that we are considering are materialized views. So, for instance, the values of the instance variables of the view object are loaded when the view is initialized. Now, suppose that one of the instance variables is (or contains) an object. An issue is whether this object should be loaded as well at initialization time. This is clearly application dependent:

1. Consider an application that allows the user to check out a document to work on it at home disconnected from the database. Then the system should load the entire document and not wait for an access to some subsections because the connection may not exist anymore when the subsection is accessed them.
2. On the other hand, suppose we adopt blindly the policy that any reachable object is loaded. In some cases, one may end up loading a large useless portion of the database in the view.

In *ActiveViews*, the default for objects occurring in or reachable from the view instance variables is *load deferred*. This default may be overruled by defining instance variables as *load immediate*. Consider, for instance, changing the mode of attribute *catalogue* in the view *Customer*:

```
load immediate catalogue : Catalogue
     has value DBcatalogue;
```

The database catalogue is then loaded entirely at creation time.

Actually, there are a number of issues related to large objects depending on whether we store loaded objects in the active view (i.e., possibly in the database since the view is an O_2 client) or in the view interface (i.e., on the machine where the view was activated):

1. Storage in the active view: when the catalogue is updated in the database, we start having several versions of this catalogue. The versioning mechanism of O_2 can be used here.
2. Storage in the interface: the interface may have to use secondary storage (files) to store its data.

Finally, wherever the loaded object is, when refreshing it, we do not want to resend the entire data to the interface. So we have to use a Δ approach, i.e, send an incremental change.

4.5 Access Rights

It should be stressed that access rights have to be much more sophisticated in electronic commerce applications than in most standard database applications. We therefore provide the means to attach an access predicate to any instance variable or method of a view. This predicate may use many information: the database or view data, the user identification, etc.

In general, access rights determine the modes of instance variables (e.g., read/write), and determine if methods are active or not. For instance, we may change the initialization process of the view *Customer* to allow the creation of the view even if the user does not have the appropriate credit. Then, the customer will be able to view only his/her past orders. This can be implemented by adding the following control instruction on all instance variables but *myorders* and all appropriate methods:

```
if (owner->approved_credit() >= 0)
```

Adding access rights related to groups of people is done a similar way. For instance, we can decide that company people can pass orders even if their credit as a customer is less than 0. For this, we change the control instruction in:

```
if (owner->approved_credit() >= 0 or owner->in_group(``employee''));
```

Remark 4.1 Access right checking may be quite expensive. In many cases, the access rights will depend only on the parameters to 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. □

Remark 4.2 Another issue here is asking OQL queries from the view. Predefined OQL queries packaged in methods can be used. The only access control will be performed by controlling the method activations. It will also be possible to provide (in a method) a window where the user will type an arbitrary OQL query that will be sent to the database, evaluated and the result returned to the view. (The result in that case may be under the form of an HTML document *à la* O_2Web.) The problem of verifying whether a given OQL query is within the scope of the specified access control for a particular view will be ignored here. □

5 Tracing the activity of monitored objects

In the previous section we showed how the application objects could be monitored using the O_2 notification mechanism. Furthermore, we showed how this was used to keep (part of) the view data fresh. Another reason to monitor objects is to keep a log of the application activity. 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 or to be able to handle eventual disputes between the participants. It usually requires logging events of higher abstraction than the basic create/update/delete events previously introduced and can also be used to write active rules in a nicer way. How the log is used is explained in the next section. In this section, we explain how it is constructed.

The application log is maintained by the active view manager and, more precisely, by its tracing module that is in charge of recording information specified by the application programmer through trace declarations. For instance:

```
trace write(o: Order) in class Customer;
trace entitle_to_promo(p: Promotion, c: Customer) in class Vendor;
trace start(source: Source, level: string) in class Alarm
```

The code generator guarantees that traced method calls will be notified to the *active view manager*. Note that it is easy to generate the code required to trace methods defined within the application. Methods that are part of the original database schema can be traced only if the programmer has provided the necessary instructions. The code generator will inform the programmer if this is not the case who can then modify the database methods accordingly or the application itself (notably by using virtual classes).

When notified of a traced method call, the tracing module records the method name, the time of the call, the method sender and, eventually, the parameters of the call in an object called *trace*. This information is stored in the following way: *trace*.C.m contains the set of calls to the traced method m of class C. Each such call is modelized by *time* and *sender* attributes plus one attribute per method parameter.

The log can be viewed as a partial history of the activity of the view application and can be queried. For instance, the following query returns the promotions that have been sent to company employees along with the vendors that have sent them.

```
select X.p, X.sender
from X in trace.Vendor.entitle_to_promo
where X.owner->in_group(''employee'')
```

The next method returns the list of origins of emergencies detected by the application:

```
select X.source
from trace.Alarm.start X
where X.level = ''emergency''
ordered by X.time
```

We now illustrate two ways of using the tracing mechanism: active rules and control of business models.

6 Rule specification

Since business models specify a protocol of exchanges among partners to a transaction, their semantics is primarily behavioral. Their specifications therefore have a strong dynamic component, reminiscent of active databases and transactional workflows [wor93].

In this section, we first introduce active rules that provide an alternative and centralized way to capture the active component of the applications. We do this by adding a triggering mechanism to O_2. This is done in a minimal way compared to some more sophisticated active rule mechanisms that have been proposed, e.g, in the context of O_2, the NAOS system [CC96]. Next, we show how a technique for controlling the behavior of the system – based on the Spocus transducers of [AVFY98] – may be introduced in our system.

6.1 Introducing active features

The *active rule module* of the *active view manager* controls the execution of some active rules. The active rules we are considering are of the form:

```
on <event>+
if <predicate-query>
do <program>
```

Events can be of two kinds: instance variables updates (captured by the notification mechanism, see Section 4.1) and traced events (see previous section).

The *active rules module* maintains the set of active rules and a table of all traced events and basic notifications that are relevant to these rules. When such an event/notification is detected, appropriate actions (eventually none) are undertaken by the module.

This is much simpler than many active mechanisms that have been considered. For instance, many systems consider an array of modes such as deferred or immediate and many events such as beginning or end of transaction, external events, etc. We decided here to focus on a simple mechanism that seems to suffice for the applications we consider.

Let us now see how active rules can be used. For this, we use the application given in Section 3 and redefine the activity of a vendor using active rules instead of *fresh* mode. Using one or another solution is often only a matter of taste: do we prefer to centralize the activity description or spread it across the application? However, since the programmer knows the semantics of the application, using active rules may be more efficient in some cases. Indeed, it may avoid recomputing the value of an instance variable that the user knows is not affected by a change, information that the system may not be intelligent enough to infer.

A vendor view has mainly two active parts: it notifies the dispatcher and its customers when it is disactivated (either definitely or temporarily) and it send promotions to customers. We assume here that the logged events we use have been correctly traced and that similar rules are defined for the *Customer* and *Dispatcher* views.

```
on sleep(), quit() in class Vendor do
   {Customer c;
    dispatcher.available_vendors -= set(sender);
    for (c in sender.customers)
       c->tutored_by_vendor(nil);}

on entitle_to_promo(p: Promotion, c: Customer) in class Vendor do
   {c.promos+=list(p);}
```

This example, illustrates the use of traced events. One can also use the active view manager queue of basic notifications. For instance, the following rules updates the *customers* application variable when a *Customer* view is created.

```
on create in class Customer do
   {customers+=set(sender);}
```

6.2 Spocus transducers

In this section, we briefly illustrate how to introduce in the framework of active views an approach to business models previously proposed by a subset of the authors, namely Spocus transducers.

In the electronic commerce context, it is particularly important that the specification of a business model be easy to understand, for various reasons: the participants in the exchange must understand each other's business models, the business models themselves can be subject to negotiation, and business model specifications may carry contractual values. Therefore, a high-level, declarative specification of business models is particularly desirable. This motivated us to focus in the Spocus transducer on simple rule-based specifications.

In the original Spocus approach, business models are formalized as follows. The state of an application is described by a relational database. The interaction from the outside world is captured by a sequence of input relations (here the clients). A client may send commands to the business model only by inserting tuples in these relations. The application responds by inserting tuples in some output relations. Thus, the model can be viewed as a machine that translates an input sequence of relations into an output sequence of relations, therefore the term *relational transducers*. In Spocus, the rules to specify the input/output translations are specified in nonrecursive datalog [AHV95] with a very weak form of negation. The goal is to be able to solve a number of important problems arising from the design and verification of business models that would be undecidable in a more general setting. Examples of such problems are the following:

- *Log validity*: testing whether a given log sequence can actually be generated by some input sequence. This problem arises for instance when, for efficiency and convenience, the application of a supplier is allowed to run on a customer's site. The trace provided by the log allows the supplier to validate the transaction carried out by the customer.
- *Goal reachability*: most business models are geared towards achieving a certain goal, such as delivering a product under certain conditions. Goal reachability checks whether a set of goals can be reached by some run of the transducer. (The system should also be able to guide the user towards achieving these goals.)
- *Temporal properties*: verification of desired temporal properties of the business model, such as "*No product can be delivered before payment is received.*"
- *Business model containment*: this concerns testing whether every valid log of one transducer is also valid for another. The main motivation for considering this question is customization of business models. Current electronic business models tend to impose on users unnecessary constraints and limitations. It would be desirable to let the users customize the basic business model for their convenience or to conform to their own regulations. Then it is necessary to verify that the new business model still conforms to the original semantics, i.e. the valid logs of the customized model remain valid in the original model.

Obviously, problems such as the above are undecidable for unrestricted applications. They are decidable (under reasonable restrictions for some) for Spocus transducers.

Our goal here is to introduce the control of business transactions provided by Spocus in the context of active views. Indeed, we would like to control as much as possible the activity with Spocus. For that, we may function at two levels:

1. Observation and notarization: Spocus simply observes the activity of the application via traces and verify that they are in agreement with the business rules as defined by a set of Spocus rules.
2. Direct control: the Spocus rules are compiled into a set of active rules and the contractual part of the business transaction is directly under the control of the Spocus module.

We will focus here on the second approach that provides tighter control.

Input tuples are captured by some traced method calls where the fields of the tuples are the arguments of the method calls. So, the firing of some method is interpreted by the Spocus module as some new input fact that is externally issued and modifies the state of

input method	$submit(herald, john)$ $submit(newsweek, susan)$ $submit(hustler, john)$	$submit(le\text{-}monde, susan)$ $pay(herald, john, 55)$ $pay(newsweek, susan, 48)$	$pay(newsweek, susan, 45)$
private method	$bill(herald, john)$ $bill(newsweek, susan)$	$deliver(herald, john)$ $bill(le\text{-}monde, susan)$	$deliver(newsweek, susan)$

Figure 4: A run of the business model

the business transaction. Note that the Spocus module does not distinguish between facts that result from user commands (e.g., issuing an order) and facts that may result from computations of the system (e.g., starting a general promotion).

Output tuples are also captured by other method calls. But to be on the safe side, it is assumed that the Spocus module is only allowed to fire these methods (they are *private* to the Spocus module). By this, we guarantee the validity of the input/output relationship controlled by Spocus.

To complete the picture, it suffices to notice that the Spocus rules are implemented as active rules, which is easy since they are by nature rather simple.

The following simplistic example illustrates a business model definition in Spocus. It uses four traced methods that are not defined here: *submit*, *pay*, *bill*, *deliver*. In the presentation, we use the syntax of active rules (instead of that of Spocus) to highlight the integration of Spocus in the system. The object controlling the Spocus module is called *spocus*.

```
input submit, pay in class Customer
output bill, deliver in class Spocus
rules
on submit(neworder : Order, owner : Person) in class Customer
   if ( neworder->available()
        and not trace.Spocus.bill(neworder,owner) )
   do spocus->bill(neworder,owner)
on pay(neworder : Order, owner : Person, amount: dollars) in class Customer
   if ( trace.Spocus.submit(neworder,owner)
        and neworder.price = amount
        and not trace.Spocus.deliver(neworder,owner) )
   do spocus->deliver(neworder,owner)
```

A possible trace of this model is shown in Figure 4 where *herald, newsweek, hustler, le-monde* are orders and *john, susan* are customers.

One could imagine developing business applications using arbitrary active rules. To conclude this section, we would like to stress the advantages of using a very limited form of control as in Spocus based on simple rules with restrictions not detailed here in particular on the use of negation (see [AVFY98]). With Spocus rules, (i) the semantics of the business model is declarative, and thus easier to explain to the participants, and (ii) we are able to reason about the program and prove properties such as specified above. We believe (and this has yet to be demonstrated with real business applications) that the Spocus model is rich enough to specify a wide range of practically significant business models.

7 Conclusion

The user may be confused at this point by all this merging of technology. The bottom line is still: how do I write an electronic commerce application or more generally some cooperative work application in this system. The following describes the main steps in specifying an active view application:

1. For each view, we first specify the data and methods in the view. This may involve restructuring some data, hiding method, adding virtual data. It also involves choosing the appropriate modes (read, write, fresh, etc.) and access rights.
2. This done, we have to eventually customize the interface or specify the presentation of the views.
3. Finally, we can specify a global control of the application activity by defining some active rules and/or Spocus rules.

Examples of applications that can benefit from active views:

Electronic sales: management of a supplier's catalogue and electronic order/billing/payment. (The management of the catalogue is a typical case where incremental updates are compulsory.)

Bug report: reporting and management of bugs in a software product. (Many needs for notification and active rules.)

Web site management: with the possibility for authorized actors to update the site selectively (e.g., their home page); to monitor access to the site and introduce push technology. (This requires for instance activity tracing.)

Data check-in/out: possibility to load some data (documents), disconnect from the database, work on them and reconnect later. (This stresses the need for sleep mode for views and update merges.)

Database access control: possibility to introduce a sophisticated access control on a database.

Acknowledgments We thank Guy Ferran, Sophie Gamerman and Arnaud Sahuguet for discussions on this work. Also, the members of the Rocquencourt EC group provided valuable comments, and in particular, Bernd Amann, Sihem Amer-Yahia and Jérôme Siméon.

References

[ACM93] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proc. VLDB*, 1993.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading-Massachusetts, 1995.

[AVFY98] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. In *ACM PODS*, 1998.

[BS81] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. on Database Systems*, 6(4):557–575, 1981.

[Cat94] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, San Mateo, California, 1994.

[CC96] T. Coupaye and C. Collet. Composite events in naos. In *Proc. DEXA'96*, 1996.

[CDSS98] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediator needs data conversion. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1998.

[FC85] 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.

[Gem] GemPlus. Smart card homepage. http://www.gemplus.com/.

[Kel82] 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.

[PV97] P. Picouet and V.Vianu. Semantics and expressiveness issues in active databases. *J. of Computer and System Sciences*, 1997. to appear.

[SAD94] C. Souza, S. Abiteboul, and C. Delobel. Virtual schemas and bases. In *Proc. EDBT, Cambridge*, 1994.

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

[WC95] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan-Kaufmann, San Francisco, California, 1995.

[wor93] Special issue on workflow and extended transaction systems. *Data Engineering Bulletin*, 16(2), 1993.

[YA96] 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.