Evolving Databases: An Application to Electronic Commerce

Bradley Fordham * Serge Abiteboul † Yelena Yesha ‡

February 10, 1997

Keywords: electronic commerce, negotiation, evolving algebra

1 Introduction

Abstract Many complex and dynamic database applications such as product modeling and negotiation monitoring require a number of features that have been adopted in semantic models and databases such as active rules, constraints, inheritance, etc. Unfortunately, each feature has largely been considered in isolation. Furthermore, in a commercial negotiation, participants staking their financial well-beings will never accept a system they cannot gain a precise behavioral understanding of. We attack these problems with a rich and extensible database model, *evolving databases*, with a clear and precise semantics based on *evolving algebras* [5]. We also briefly describe a prototype implementation of the model [12].

The first contribution of this paper is a rich and extensible database model primarily aimed at capturing rapidly changing environments. We describe electronic commerce *negotiation* using this *evolving database* (EDB[1]) that captures the state of traded products, of negotiators, and the accepted laws governing the particular negotiation. We use the term *evolving* to stress the extremely dynamic nature of a negotiation. Negotiators should be able to change the product descriptions, orders, and even protocols of negotiation on the fly (e.g., by introducing a mediator in case of conflict). Although we will focus the presentation in this paper on the singularly interesting domain of commercial negotiations, the model is clearly not limited to such applications.

An EDB is built using extensible sets of *domain features* (DFs), and entities that are described by instances of these domain features. For instance, *constraint* could be a domain feature and *salary* $>$ *100K* an instance for this feature. Some of the DFs are quite generic capturing: attribute/value pairs (as in a product description), relationships between these entities, active rules (e.g., to modify a quoted price when the product description changes),

*NIST; CS Department, UMBC

†CS Department, Stanford University. On leave from INRIA-Rocquencourt, France

‡NASA-CESDIS; CS Department, UMBC; with support from IBM CAS and Johns Hopkins University

[1] EDB – not to be confused with "extensional database"

simple constraints (e.g., guaranteeing a profit margin), etc. Other DFs may be much more specific such as a DF for authentication or confidentiality or a DF to describe the compatibility of various components. DFs will therefore constitute the EDB meta-data for the particular application.

Three major issues are: combining all DFs in a coherent system, handling changes of arbitrary nature, and providing semantic clarity. We resolve these difficulties with our rich and extensible EDB model, derived from the semantically formal mathematical basis of *evolving algebras* (EAs) [5]. In some sense, EAs serve as formal foundation for the very complex and dynamic evolving databases in much the same way as relational calculus serves as a formal foundation for the beautifully simple but largely static relational databases.

We may now make more precise the concepts of domain feature and entity. A DF is specified by (i) a syntax, the set of correct formats for instances of this particular DF, and (ii) a semantics, the translation of the terms in that syntax to rules in the underlying EA. ¿From this, it becomes clear that the conversions between the EDB and the EA worlds are central to our approach. Roughly speaking, the user sees and manipulates EDB entities, and the engine "executes" EA rules. The state of a negotiation changes because of EDB changes requested by humans, and EA changes performed by the EA engine; changes are propagated back and forth between the EDB and the EA worlds.

Our second contribution is the development of a prototype system, an environment to support experimentation with the evolving database model. The core of our system consists of the EDB query engine, the EA engine, and the converters between EDB entities and EA rules. These are agent programs written in C++. The prototype is built on top of Postgres95 which provides for concurrency control, persistent storage, recovery, etc. This database back-end is used primarily to store the DF meta-data, entities (sets of instantiated DFs), the EA rules, and the state of the EA. Version 1 of the system is currently running. We are rapidly improving it.

Our last contribution (still very limited) is the use of this model in the context of EC negotiations. Negotiation modeling has been so far the guiding application for this work. Some toy negotiations have already been implemented. We intend to turn now to *real* EC applications.

We have made a careful study of what we term *semantically complex* UoDs, those applications where the semantics surrounding data is at least as critical as the data itself. An example of this would be EC negotiations where the attribute DF, a simple k tuple of attribute/value pairs is not enough to model the real world. Here there is interest in the time DF (when values change), the constraint DF (ensuring that certain add-ons which require additional components are not added in isolation), the compatibility DF (ensuring that all components of a system can effectively interconnect), the behavior DF (perhaps defined to map inputs to a product onto outputs in a given domain), complex relationships (one product replaces an earlier model or alternatively is functionally equivalent to it). Here the usefulness of the simple attribute-value vector pales in the significance of other semantic understandings, i.e. other DFs, in the real world being captured. For contrast, a semantically uncomplex UoD would be an employee/project application where the attribute DF, simple k-tuple modifications, is sufficient. The term UoD is often defined as the data+semantics of a real world. [2] We are interested in those real worlds stressing the semantics. The pre-

[2]We would argue that data is simply one DF, the attribute DF, out of many which define any givien

dominance of our efforts have focused on electronic commerce and commercial negotiations since we see this as one of the most semantically complex UoDs as well as being a problem in need of timely solutions.

We ask ourselves now, what it is that semantically complex UoDs demand. They demand flexibility. Entities in a semantically complex system evolve over time, in response to stimuli external to the system, in response to the other entities that surround them, etc. Take negotiators, they may get more "desperate" to deal as time progresses. They may have an external sentience supply them suddenly with new information such as a newsflash relevant to the negotiation at hand. They may react differently as new vendor and buyer agents enter the fray. The next question is how do they evolve. The DF set may change, e.g. suddenly one party wishes to place a special provision onto the purchase contract requiring the addition of a constraint DF for validation purposes. The DF instances that define entities may change, e.g. adding a new attribute to a product or modifying its stated behavior. New operations over DFs may need to be created as internal or external functions, e.g. performing the transitive closure over input-output specifications to ascertain the input/output of complex systems. Access rights may change, e.g. originally permitting an entity to see the price only after all other terms of the contract have been agreed to, and later altering that strategy in an attempt to conceal price completely from other vendors. Even basic understandings such as *true* and *false* may need to change continuously as fuzzy logic schemes become natural, e.g. moving from does a given product meet my requirements to which product best meets my requirements.

In database terms we find all of the following to be potential hinderances in the effective modeling of semantically complex UoDs.

Rigid Schemas: Force a notion of classes or types and discouraging instance variability How many PCs really have the same structure?

Rigid Data Models: Force the domain expert into small, fixed set of primitives and operations that may be totally unnatural to his/her real world How many contracts or negotiation strategies naturally look like relations, complex objects, trees, ...?

Simple Notions of Scope/Granularity: Force concurrency control, access control, query language, and other mechanisms to access data only in chunks easily definable by the model (records, objects, ...) How many negotiators reveal in all-or-nothing fashion their entire understanding of the contract, the service, or the opposing negotiator or for that matter all of any sub-object of any of those entities? Why wouldn't it be natural for a vendor to give me the first 2 digits of the price while holding back the last 3 until a later time?

Inability to Combine Semantics: Force domain modelers to choose a deductive, active, object-oriented, constraint, or other some other DF in addition to the attribute DF. Each of these offers a certain advantage in terms of semantic capture and naturalness of specification, but no system we are aware of simultaneously provides them all. Why wouldn't a product most naturally be expressed in terms of attributes, constraints, legal logical inferences, behaviors, and other DFs than it could be expressed with only

UoD. In our experience, most real worlds rely on DFs beyond this.

one or two DFs? In fact, there are semantics that cannot be recast from a specification in any given DF to an equivalent representation in another DF.

Prohibitions Against Runtime Evolution: Force real-worlds to hold structure, or DFs, or DF instances, or sorts, or entities, or entity-types constant over time in spite of all observed stimuli. How many UoDs should really hold anything absolutely constant as the world around it in which its creators live continuously evolves? Why are conventional database being constantly restructured and extended with features like triggers and stored procedures?

The paper is organized as follows. Section 2 is a brief glance at EAs. In Section 3, we describe EDBs introducing the notion of DFs and entities, and their translation to EA rules. A short description of the system architecture is given in Section 4. The application to the modeling of negotiations in electronic commerce is given in Section 5. The last section is a conclusion.

2 Evolving Algebras, Overview

We only briefly describe evolving algebras here insisting on their particular interest in our setting. For more information, the reader is referred to [5].

Evolving algebras (EAs) intend to bridge the gap between specification and computation by providing more versatile Turing-complete machines. EAs allow one to simulate algorithms at a variable level of abstraction, thereby making easier the validation of the mathematical model against reality [16, 18]. Evolving algebras were used to obtain specifications of a number of complex systems. For instance, a Prolog machine is specified as an EA in [6]. In the database field, evolving algebras were used for specifying the semantics of object-oriented databases [15] and active databases [31].

EAs form a variant of first-order logic with equality. The *vocabulary* of an EA is a finite collection of fixed-arity *function names*. Functions are in general partial (i.e., they may return the particular value $undef$) [17]. They may be *internal*, fully under the control of the EA, or *external*, black-boxes to the EA. They can be *static* (i.e., never reinterpreted), or *dynamic* (possibly acquiring new semantics during an execution). *Relations* are also captured as functions into $\{true, false\}$. Particular sorts (*universes* in the EA terminology) can be partitioned out of the *super-universe* using unary relations.

A *static algebra* or *state* is the super-universe together with *interpretations* for all the functions. The state changes because rules (*guarded updates* in the EA terminology) fire when a particular predicate, the *guard*, is satisfied. In a given state, a set of updates is triggered which leads to the next state. An EA *program* is defined by a set of rules and a *run* by applying the rules to an initial state.

Another important aspect of the applications we consider is *distribution*. Conveniently, distributed EAs [18, 29] were proposed for such contexts. In the EA world, an *update family* is an unordered collection of update sets. A *distributed EA* is an update family where each update set is envisioned as an agent of the distributed computation. In a distributed EA, each agent is identified by an element in the common carrier, it's *name*, and possesses a local program, vocabulary, and state-view. Agents communicate by sharing locations through common carrier elements. An agent *moves* by firing its program at it's view of the state in the usual way and can be *teamed* with other agents to provide cooperative action.

3 Evolving Databases

In this section, we discuss the need for EDBs and present them using some simple examples demonstrating the translation to EA rules. Our goal is to faithfully capture richer data semantics in a constantly changing environment. Object-oriented [4, 10], active [8, 26], deductive [13], functional [32], semantic data models [19, 27], graph based [9, 3, 24], temporal [7, 14, 22, 28], spatial [23, 11, 28], and many other database technologies all have as their primary focus the layering of more human semantics over the traditional symbolic representations of data. The effects of these efforts, in terms of expressive power, have been impressive. But the quest for more semantic capture has not stopped. Another major problem is that we do not have a clean way to combine these different concepts in a common data model. We propose EDBs as a solution to this problem since they cleanly integrate standard semantics while remaining open to the introduction of novel DFs.

Consider the modeling of PCs. In terms of data we must capture attributes and values describing products and components (modems, cd players, etc.), as well as the classifications of and relationships among them. These are common tasks of database systems. Critical are domain-generic semantics including: constraints on values, context-sensitive rules, and deductive mechanisms. But also important are domain-specific features since a data model should provide *natural* ways of describing the UoD including, for examples: interface compatibilites, sensitivities to environmental conditions, and behaviors. On the other hand, a data model must provide the system with an effective way of performing data manipulation operations. EDBs meet both criteria. They provide a general and natural paradigm for domain modelers thinking in terms of whatever DFs that they consider useful every day. Also, this extensible and flexible EDB formalism can be converted to an EA that is readily executable by machines.

The model is based on *DFs* and *entities* defined as instances of these DFs. The basic operations are the insertion, deletion and modification of DFs, and the creation, deletion and modification of entities. We must have the means to specify some new sorts or universes (e.g., color) or modify existing ones to adapt the UoD to the application. We must also be able to translate the syntax (notation) of an entity to its formal semantics (denotation) as EA rules. Although not fully detailed here, we also have in the model ways of expressing relationships such as *isa* and *hasa* to simplify the specification task by code reusability.

Suppose we have this extremely flexible and extensible entity model where everything from DFs to entities is dynamic. What we still lack is a way to satisfactorily specify the semantics of this EDB and realize it inside of a DBMS. The primary difficulty is clear. How can diverse DFs be meaningfully combined? In particular, what does it mean to mix the firing of rules, drawing of inferences, validation of constraints, updates to attribute/values, invocations of methods, operations over behaviors, and any other actions generated by some DF? We must find a way to reduce arbitrary DFs to some lowest common denominator which can be meaningfully "executed" by the system and "experienced" by the system-observer. Our solution reduces all semantics for DFs to EA rules by relying on the specification/computation bridge and the variable level of abstraction EAs afford. We seek to prove with this research that this evolving algebra can be interacted with and experienced by users in ways which are much more interesting than common interactions with conventional databases and query languages.

Entities, attributes and values

Our approach is *name*-driven. Each entity is represented by a name that uniquely identifies it, and in that sense can be seen as a logical object identifier. This is accomplished in the EA formalism with 0-ary distinguished functions representing well-known concepts in the UoD. A particular universe, *Entity*, contains all names of entities identifiable by the EDB. The PC product application may have the names $PC1()$, $PC8000()$, $Mr.Jones()$, *PaloAltoOffice*(), $Modem87()$, etc. This universe can be modified easily by introducing new names or removing obsolete ones.

Based on that, it is easy to associate a value to an attribute within an entity. We simply create a DF, call it *attribute* and supply a syntax, ?*attribute* =?*value*, along with a semantics, i.e. the translation into EA rules, ?*attribute*(?*entity*) :=?*value*. (Here the tokens preceded by "?" are variables.) Thus, a simple unary function of the EA taking the name of the entity as an argument and mapping it to the current value of the attribute is sufficient to capture the *attribute* DF. At this stage, we can already define simple entities corresponding to tuples in a relation. Also attributes can be assigned new values in the standard EA way, by updating the corresponding attribute-function, e.g:

```
manufacturer(PC1()) := PCsRus()
color(PC1()) := green()
```

The universe PC could be defined either extensionally or intensionally. (Indeed, the EA would even allow PC to be defined externally as a black box that decides if an entity is a PC.) In all cases, this comes down to defining a function PC : *Entity* $\rightarrow$ *bool* which returns *true* if the entity is a PC. Domain modelers may further sub-classify entities by creating sub-universes of the *Entity* universe in appropriate ways.

Constraint

We now consider simple constraints which will allow us to specify our first non-trivial DF. Not surprisingly, the EDB rule approach will tend to favor a more active view of constraints than standard integrity constraints.

To specify a constraint DF, we need first to give the legal syntaxes for use in the EDB:

```
?val(?entity) between ?a:Num and ?b:Num          ?val(?entity) in ?universe
```

EA functions are inherently untyped. Thus the syntax (: $Universe$) is interpreted as a constraint, i.e. that the value must be chosen from the universe specified. This constraint may be omitted on any variable that the domain modeler sees fit[3]. Next, for each syntax, we specify the EA rules that define its semantics by stating what to look for in the EA state (the guard) and what to do in response (the update). The following EA rule schemata could be attached to the above two syntaxes:

[3] Certainly, a function could simply return $undef$ if its arguments were somehow not acceptable to the computation.

if ?val(?entity) $<$ *?a* $\vee$ *?val(?entity)* $>$ *?b then ?val(?entity)* $:= (?a+?b)/2$
if $\neg$ *?universe(?val(?entity)) then ?val(?entity)* $:=$ *undef*

The constraint domain may now be used in the specification of entities. For instance, the EDB specification of a PC could be:

```
Define Entity 'PC8000':
       Attribute:  color=green, price=1499
       Constraint:  color in COLORS, weight between 20 and 25
```

We can specify arbitrarily complex syntaxes and a semantics that are as rich as needed since EAs are Turing complete. In spite of this expressive power, the use of EAs results in more readable code since the EAs can be focused to the appropriate level of abstraction.

Active rules

Since EAs are themselves based on active rules, it is rather simple to introduce rules into our EDB. We illustrate here a simple example where the distance between the active rule in the EDB and the rule in the EA is very small. However, in general, this does not have to be the case. The syntax of the EDB rules will actually be much simpler than the general syntax of the EA, so that triggering can be achieved efficiently. At the level of the active rules, we do not need any kind of Turing completeness. Furthermore, the control of the EDB rules (e.g., when are the rules triggered) may be specific to an application and does not have to follow the control of the EA rules. So, for instance, one could specify that the active rules trigger only at commit time, or at method activation time. This separation between active rule control and database engine control which is a source of many difficulties in active databases disappears here because of the clean separation between the active rules (in the EDB world) and the system rules (in the EA world).

We can introduce a DF called *egu* (event-guard-update), enumerate the legal syntaxes, and define the interpretation for each syntax as a rule schema. We may accept the following syntax:

```
on ?event if ?guard do ?update
```

with the natural translation to the EA:

if Detected(?event, ?entity) $\wedge$ *?guard then ?update*

This uses a predicate of the EA called *Detected* which takes as its arguments the event being monitored and the entity over which that monitoring is scoped. A sample event is the modification of the state of an entity. We should question the exact meaning of the function *Detected.* What exactly is it within our EA? It would be possible (but tedious) to define it internally within the EA. It seems simpler to view it as an external function provided by the system. Its semantics should be clear enough to avoid having to be formally specified in the EA. We also must define in the underlying EA a universe called *Event* with one distinguished element for each detectable event.

Specification Sharing via Relationships

Critical to our EDB paradigm is the capture of the semantics found in the sharing of specification between entities that arises as entities are related to each other in theoretically infinite numbers of ways. This, as all data and semantics of our EDB, is captured as a DF. To define this *relationship* DF, the domain modeler must simply create a syntax for each relationship type (i.e. mode of specification sharing) and define for each a semantics. As one example, it may be natural to specify connectable-ness and connected-ness between PC components. The EDB paradigm is flexible enough to support such domain-centric relationships. The domain modeler might be comfortable with the following syntax which may naturally occur in the PC UoD:

```
connected(?connector-id1,?connector-id2)
```

where the understanding is that two connectors, ?connector-id1 and ?connector-id2 are currently cross-connected. We now semantically interpret this relationship syntax via EA rules. We can trivially capture the fact that a connected relationship exists for this entity, and therefore the connected semantics should be enforced, by setting true an EA relation of the following form:

connected(?connector-id1,?connector-id2):ConnectID $\times$ *ConnectID* $\rightarrow$ *Bool*

We can further add a rule to guarantee that the two connectors are of the same type and differing sex:

if connected(?connector-id1,?connector-id2) $\wedge$
(type(?connector-id1) $\neq$ *type(?connector-id2)* $\vee$ *sex(?connector-id1)=sex(?connector-id2))*
then connected(?connector-id1,?connector-id2):=false

This is the semantics of our connected relationship. We observe that this rule of the EA has a truth-maintenance flavor. If the sexes or the types of connectors that are connected become incompatible then the EA will, on the next execution step, correct the problem.

More standard types of relationships are easily supportable between entities as well, e.g., ISA(e_i,e_j). Most simply, we could define the ISA relationship to ensure that the sub-entity always contains all of the values instantiating the DFs of the super-entity in each of its DFs. We expect the semantics for most relationships to be given as global rules that define the meaning of that relationship type for all entities. However, in the usual hyper-flexible and hyper-dynamic mode of EAs, the scope of these rules can be arbitrarily restricted. For example, we could define the HASA relationship as a global rule with different semantics for different combinations of sorts, e.g:

HASA($e_i : buyer$, $e_j : product$)
HASA($e_i : vendor$, $e_j : product$)

In this scenario, vendors having products can increment the currently available stock values while buyers having products can "check-off" some of the needs that were expressed in their purchasing strategy.

Application Specific, Natural DFs

Recall that our over-arching concerns are naturalness of specification and clarity of operational semantics for all entities in a very complex and dynamic UoD. This implies a need to allow domain experts to define and experience the effects of very domain-specific features when the generic ones such as constraints do not fit closely enough with their intuitions about the real world. EDBs achieve this goal. As aforementioned, EAs are a Turing complete specification/validation methodology. This implies two things. Any DF should be representable and all DFs that were previously thought of in passive terms will now seemingly become more active as they are reduced to their operational semantics. A domain modeler should never have to abandon the EDB paradigm itself to capture a new DF present in his/her dynamic, and semantically complex entities.

To conclude this section, we want to stress the flexibility of our approach. New DFs can be added on-line to enrich the EDB. These DFs may contain semantics defined by very complex modules. For these modules, we have the choice of specifying them directly in the EA or introducing them as externals. Indeed, this may depend on the level of abstraction that is being considered; the same function may be viewed as external and later refined as explicit in the algebra if the domain modeler wants to be convinced that it is indeed behaving properly. Such DFs with external functions may prove invaluable in particular to "open" the system. One may consider for instance integrating in this manner: access to a foreign query language (e.g., SQL or OQL), deductive capabilities, access to external formats (loader/printer of EDI or HTML), communication protocols with participants, distributed supports such as CORBA, authentication mechanisms, etc. Electronic commerce applications will require such support, and our EDB approach allows us to provide it within a coherent framework.

4 A System Architecture

Figure 1 provides a high-level overview of our system architecture. The system is architected as a community of C++ agents. This means that direct communication from any module/agent to any other is possible. We see this as a clear advantage in permitting flexibility of operation. Figure 2 shows a representative collection of our system agents and a few of the more dominant communication links between them. The agents are responsible for the interfaces (in particular, the query languages), the EA engine, the conversion engines, and the repositories. We now briefly overview these main functions.

The EA manager and engine. This can be viewed as a very flexible and dynamic active database component. The EA manager is responsible for maintaining the EA state and storing its history using a backend database system. The EA engine's role is to detect rules that can be fired, interact with external code (for external functions), and fire these rules thereby updating the EA store. The daemon running the EA has direct access to the rules of the EA thereby allowing these rules and/or the execution strategies applied to them to evolve over time (e.g. modifying guards, skipping certain rules at an execution step, or enforcing a given rule-execution order). This is a form of reflective programming.

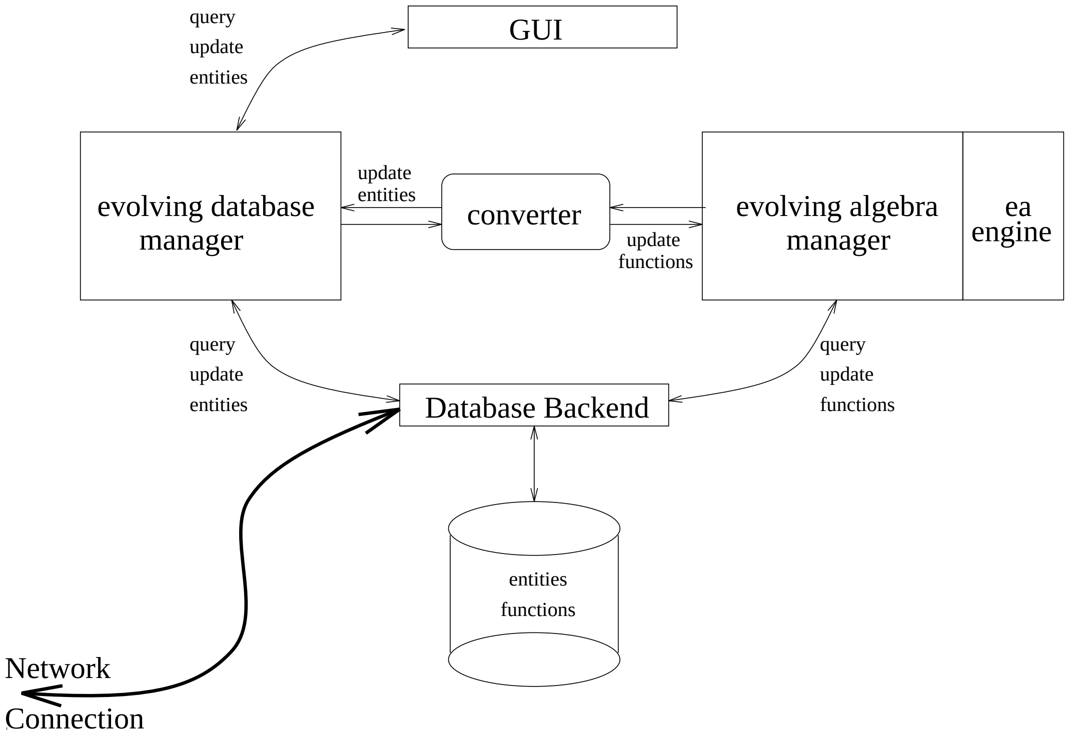

Figure 1: The System Architecture

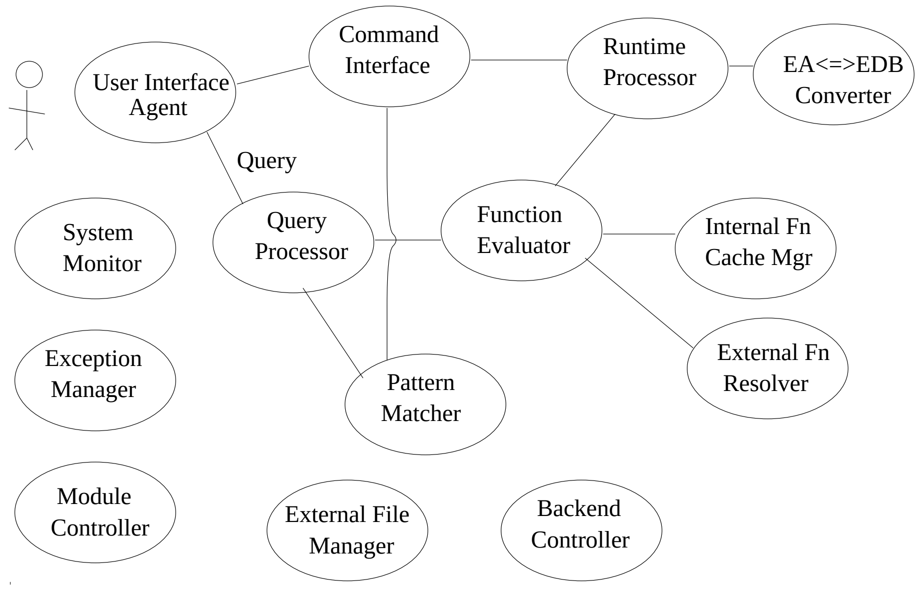

Figure 2: Dominant System Agents

The EDB manager and the converter. The EDB manager is responsible for maintaining the EDB state, following requests from users and application programs, and propagating updates issued by the EA engine. Logically, our system therefore maintains two repositories of information. The first, the evolving database, contains passive representations of DFs and entities. The second, the evolving algebra, contains an active description of the universe of discourse captured as EA functions and rules. An essential aspect of our system is the maintenance of the semantic integrity between these two worlds with data and code being constantly converted and exchanged between them (as a result of user requests or EA engine computations).

Loading. Consider again an electronic commerce application. In a typical scenario, a buyer could use a (distributed) query mechanism available in the EDB to retrieve products and vendors. These initial models of products and vendors found on the network are then loaded into the EDB (and migrated on user demand to the EA after conversion). Discovery of data over the network and loading of new models are thus important functions of the system. This should also involve conversion and loading of data from various Internet sources in various exchange formats such as HTML, SGML, and EDI. This aspect will be ignored here.

Interface. Perhaps the most complex component of the system is the interface. Besides using standard functionalities such as browsing or editing the EDB, the human observer(s) may, for instance, wish to control the migration of entities between the EDB and EA stores or search for more EDB material over the Internet. We have implemented and are extending an advanced SQL-like language for interacting with the EDB. Also, we are taking advantage of modern GUI technology to display entities with complex inter-relationships along with their EA-function interpretations in an understandable manner. This is on-going work. It is not yet totally clear just what the EDB language(s) should ideally be or even more basically what types of interactions users may desire. Also, because of the complexity of EAs, analysis tools have to be provided: tools to allow the simulation of EA runs (in the style of "what would happen if ...") or tools to guide users in their work (in the style of "what could be my next update?").

Database backend. The database backend stores the entire history of the UoD, more precisely, an *evolving algebra* describing this history. It provides secondary storage for both the EA and the EDB, as well as standard database functionalities such as concurrency control and recovery. We could have chosen any relational or object DBMS providing such functionalities. But we chose to use Postgres95 because it offers certain flexibilities that may prove useful as this work progresses.

5 Enabling EC Negotiations

In this section, we look at negotiations in electronic commerce in more detail and highlight the particulars of our EDB and the advantages of our approach in this context.

The domain of negotiations is very wide. (See, e.g. [33, 35, 34], for some general presentations.) We are not concerned here with issues such as security (more related to cryptography, e.g. [30, 25, 21]), or in designing complex strategies (perhaps from game theory). As we will see briefly, the EDB approach allows us to model complex electronic negotiations and the distributed EA approach allows us to to enact such negotiations.

The "static" aspect of negotiations such as product descriptions or constraints can be easily captured by EDBs, but that dimension of the application could also be captured by more standard database systems. What we additionally contribute is extreme power for: (i) rigorous and clear semantic understandings, (ii) distribution and (iii) very flexible evolution. We do this by taking full advantage of the EA formalism. This is discussed next.

Distribution and agents

Typically, an EA for a negotiation will involve several *agents*, the various participants. In this sense, we are considering *distributed* evolving algebras. Each agent has its own view of the state and its own program. (See Figure 3.) Agents locate one another by inspecting elements of the *Entity* universe found in their state-view. Once located, a request for attribute-value information can proceed via an attribute function invocation with the newly found target-agent's name as the argument. Note that the universe of agents is not fixed. For instance, a buyer may start an EA where he/she is the only agent and then later discover companies selling products of interest whose agents are inserted into the EA. Note also that a lot of flexibility is expected in terms of having these agents cooperate. This includes mechanisms for knowledge-sharing between agents or between the agent and the common EA.

Our first experiments with the EDB within the scope of EC indicate that the *evolving* side is also essential. For instance, an electronic salesperson may come equipped with several strategies. In a "pushy-salesperson-strategy", it constantly pressures the potential buyer. If this strategy fails or is rejected by the buyer, the vendor may switch to a "helpful-salesperson-strategy" only responding to the buyer's precise requests for assistance. Modes of negotiation, strategies, access controls, and many other aspects of a negotiation change constantly during a particular session. A very "static" approach to negotiation is bound to severe limitations resulting in the kind of constraining and often frustrating electronic vendors more and more common on the Web today.

We believe that our EDB/EA system solves many of these issues, in particular since the system permits agents to modify the common rules as well as their own programs, i.e. *self-modification.*

Negotiation modes

The EDB approach allows one to describe and evolve a wide variety of negotiation modes. This is required since a negotiation may be governed by fundamentally different rules depending on whether it is, for instance, a public auction (further depending on the style of the auction), the purchase of a house (further depending on the role of a potential real-estate broker), the purchase of a PC (further depending on the number of buyers and sellers involved), etc. Further, a single and forever binding initial choice of mode is often too restrictive. The participants may need to modify the negotiation mode due to a change in their perception of need or interest, legal responsibility, time-pressure, etc. All of this fits naturally into the EDB paradigm.

Participating in a negotiation

There are laws governing any negotiation, and *participants*, domain experts that will represent and validate particular negotiation modes, have to understand these laws as well as what is going on in the negotiation state. All participants must understand the basics of an EDB. This is because a participant may decide not to trust some portion of the state and therefore inspect entities on its own or perform private computations for independent verification. (After all, humans do check restaurant bills.) This is the reason for well-known universes of the EDB. All agents should understand how to inspect the *Entity* universe to access all available entities, the *Universe* universe to access all available sorts, the *Attribute* universe to access all defined attributes, etc. Clearly, a mode of negotiations entirely based on not-trusting the partners may be highly inefficient (since all agents would be constantly testing large numbers of elements in the superuniverse) and so one may decide to switch to a more trust-based negotiation mode with privileged partners. The EDB makes no a-priori assumption on the autonomy of the agents, nor on their honesty. This has to be decided on an application basis.

Strategies

The EDB approach also allows one to describe arbitrary strategies. Although it does not address the issue of the choice of a strategy (more the domain of game theory), it does enable the design and implementation of complex strategies or of tools to help human agents select a strategy. Furthermore, by its evolving nature, it allows the proposal of several strategies and even permits an agent to change strategy during a negotiation. This is critical in commercial negotiations as the marketing strategies of vendors and the purchasing strategies of buyers are notoriously capricious even if natural to the humans that are involved. Perhaps a particular buyer, for example, finds it natural to use a particular function *contentment* and then include in his/her strategy a set of rules that trigger some purchase if the function passes a certain threshold. Another buyer may prefer a theorem-proving approach to decide whether to purchase or not (by proving/disproving that it meets his/her needs).

Switching to vendors, one may decide that the best strategy to sell PCs is to reduce the price. Another vender may decide to display a wide range of colors. Yet another may use intense advertising (classifying his/her products in universes named *Quality* or *Cheap*). The possibilities are numerous. An issue in this setting, is to which extent the strategy of each participant is "official", i.e., accessible to other participants. This is leading to the next issue, namely access control.

Flexible and Evolving Access Control

Certainly, the most general setting that allows all agents to inspect/update all values in the system may be disastrous in a number of cases. Access control mechanisms are required. But in the EDB we seek not a canonical set of access rules but rather a varied collection of dynamic mechanisms whereby accessibility can be specified in a manner natural to the

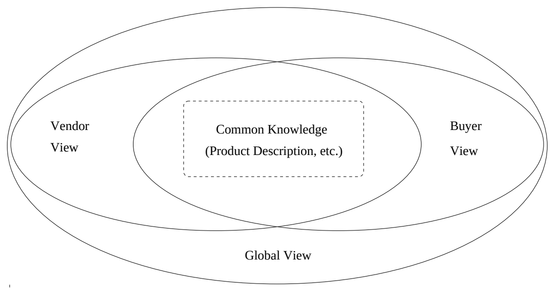

Figure 3: Scoping distributed EA agent Views

UoD and later evolved. As shown in Figure 3, each participant has a proper view of the negotiation. A common view may also be used to described the status of the current negotiation, i.e., what has been agreed upon. We seek to flexibly scope the overlap in the views of each participant.

An access rights mechanism naturally translates to adding access control guards to EA rules that will depend on the state and the identity of requesting agents. As one example, one could append to all guards a database predicate that specifies the access rights of each agent. These access control predicates can be as complex as desired although for clarity and verification reasons simplicity is often preferable.

We argue for access control flexibility with an example. Consider a buyer that is searching for a product, p, such that $color(p) = ForestGreen$. It may be the case that a vendor has created a generic product model which can be sold in a large number of colors, but initially the color of that product entity is assigned $FireEngineRed$. The vendor can inspect the rules of the buyer, thereby understanding the buyer's color desire and adapting the product model by setting the color to $ForestGreen$. In such a case, we can imagine that the buyer would not be averse to this minor invasion of his/her privacy since this is a normal interaction between a human customer and a human salesperson. The customer indicates that it is unfortunate that the product is not $ForestGreen$ and the salesperson pulls one out from behind the counter. These inspections of certain rules of other agents can conceivably go a long way to providing progress in commercial transactions, i.e. progress towards a final buy/sell agreement. Of course there are also cases where the buyer wants to keep private a certain subset of its rules until propitious moments, or even cases where this privacy is required by law. Returning to our example, consider the case where high demand for a "hot" color may increase the price. The buyer now wishes to conceal the fact that he/she is seeking this coveted product attribute/value pair.

Finally, some applications may require even more complex access mechanisms such as agent-teaming [18]. In this paradigm, neither the vendor nor the buyer can update the product model. However, another entity, the vendor/buyer team, can alter the product model in order to change the price, delivery date, etc. The team is given full access to all three entities. Also, the participants may agree beforehand that team decisions are binding, i.e. that such decisions cannot be withdrawn. Certainly there are a variety of other variations on access control available to us in the EDB/EA. We shall not discuss them here.

6 Conclusion

We believe that the main advantage of our approach is in the combination of an EDB model and underlying EAs. The EDB model with domain features consisting of a syntax and a specification in terms of EA rules, offers a very rich, extensible model. EAs provide a proper formal foundation, a requirement in the EC context. Moreover, no limitations are imposed on changes. Everything from entities to DFs is allowed to freely evolve in the general case. This makes the combination EDB/EA particularly suited to EC applications, as well as in general, suited to any complex application where rapid and large-scale evolution of state and program is essential.

¿From a system viewpoint, we continue enriching our EDB kernel. We are also improving the user interface, and performance. The expressive power of the evolving database clearly comes at some price in terms of performance. We endeavor now to minimize that price. This is a long In its current state, our system already quite satisfactorially enables experimentation with the EDB and EC applications.

References

[1] S. Abiteboul, S. Grumbach, *A Rule-Based Language with Functions and Sets*, **ACM Transac. on Database Systems**, 16(1), 1191, pp. 1-30.

[2] Nabil R. Adam, Bradley S. Fordham, Yelena Yesha, *Some Key Issues in Database Systems in a Digital Library Setting*, **Digital Libraries: Current Issues, Digital Libraries Workshop**, Newark, NJ, USA, May 19-20, 1994, Vol. 916, Springer 1995, pp. 9-19.

[3] B. Amann, M. Scholl, *Gram: a graph data model and query language*, **Proceeding of the ACM Conference on Hypertext**, ACM, Dec. 1992, pp. 201-211.

[4] F. Bancilhon, C. Delobel, P. Kanellakis, **Building an Object-Oriented Database System: The Story of O_2**, Morgan Kaufmann, San Mateo, California, 1992.

[5] E. Börger, *Annotated Bibliography on Evolving Algebras*, **Specification and Validation Methods**, Ed. E. Börger, Oxford University Press, 1994.

[6] E. Börger and D. Rosenzweig. *A formal specification of Prolog by tree algebras*, **Information Technology Interfaces**, V. Čerić, V. Dobrić, V. Lužar, and R. Paul, editors, University Computing Centre, Zagreb, 1991, pp. 513-518.

[7] J. Chomicki. *Temporal query languages: a survey*, **Lecture Notes in Computer Science**, volume 827, Springer-Verlag, 1994, pp. 506-534.

[8] S. Ceri, J. Widom, **Active database systems**, Morgan Kaufmann, 1996.

[9] M. Consens, A. Mendelzon, *GraphLog: a Visual Formalism for Real Life Recursion*, **PODS Conf.**, pp. 404-416, 1990.

[10] K. R. Dittrich, *Object-oriented data model concepts*, **Advances in Object-Oriented Database Systems. Proceedings of the NATO Advanced Study Institute**, p. 29-45, Aug. 1993.

[11] U. Finke, K. Hinrichs, *A spatial data model and a topological sweep algorithm for map overlay*, **Advances in Spatial Databases. Third International Symposium, SSD '93 Proceedings**, p. 162-177, Springer-Verlag, June 1993.

[12] Bradley Fordham, Serge Abiteboul, Yelena Yesha, *An Evolving Database Implementation*, **NIST Technical Report**, to be released.

[13] H. Gallaire, J. Minker, and J.M. Nicolas, *Logic and Databases: A Deductive Approach*, **ACM Computing Surveys**, 16(2):153-185, 1984.

[14] S. Ginsburg, **Temporal databases - theory, design, and implementation**, Eds. A.U. Tansel etal, The Benjamin/Cummings Publishing Company, Inc., 1193, pp. 272-293.

[15] Georg Gottlob, Gerti Kappel, and Michael Schrefl, *Semantics of Object-Oriented Data Models: The Evolving Algebra Approach*, **LNCS 504: Next Generation Information Systems Technology**, Springer-Verlag, 1991, pp. 144-160.

[16] Yuri Gurevich, *Logic and the Challenge of Computer Science*, **Current Trends in Theoretical Computer Science**, Ed. E. Börger, Computer Science Press, 1988, pp. 1-57.

[17] Yuri Gurevich, *Evolving Algebras: An Attempt to Discover Semantics*, **Current Trends in Theoretical Computer Science**, Eds. G. Rozenburg and A. Salomaa, World Scientific, 1993, pp. 266-292.

[18] Yuri Gurevich, *Evolving Algebras 1993: Lipari Guide*, **Specification and Validation Methods**, Ed. E. Börger, Oxford University Press, 1994.

[19] R. Hull, R. King, *Semantic Database Modeling : Surveys, Applications and Research Issues*, **ACM Computing Surveys**, Vol 19, Sept 1987.

[20] T. Imielinski, S. Naqvi, *Explicit Control of Logic Programs through Rule Algebra*, **ACM PODS Conference**, 1988, pp. 103-116.

[21] R. M. Kailar, *Reasoning about accountability in protocols for electronic commerce*, **Proceedings. 1995 IEEE Symposium on Security and Privacy**, May 1995, pp. 236-250.

[22] W. S. Kim, D. C. Chang, T. Y. Lim, and Y. H. Shin, *Temporal object-oriented data model for the schema modification*, **Database Systems for Advanced Applications '95. Proceedings of the Fourth International Conference on Database Systems for Advanced Applications**, April 1995, pp. 422-429.

[23] Y. Kornatzky, S. E. Shimony, *A probabilistic spatial data model*, **Database and Expert Systems Applications. 4th International Conference, DEXA '93 Proceedings**, Springer-Verlag, Sept. 1993, pp. 337-348.

[24] M. Levene, A. Poulovassiliss, K. Benkerimi, S. Schwartz, E. Tuv , *Implementation of a graph-based data model for complex objects*, **SIGMOD Record**, Vol: 22, Iss: 4, Dec 1993, pp. 26-31.

[25] C. R. Merrill, *Monogamous, promiscuous, and polygamous models of cryptographic electronic commerce* **EDI Law Review**. Vol: 2, Iss: 2, 1995, pp. 107-116.

[26] N. W. Paton, J. Campin, A.A.A. Fernandes, and M. H. Williams, *Formal Specification Of Active Database Functionality: A Survey*, **Lecture Notes in Computer Science**, Eds. Timos Sellis Springer-Verlag, vol. 985, 1985.

[27] J. Peckham, F. Maryanski, *Semantic Data Models*, **ACM Computing Surveys**, Vol. 20, Sept 1988.

[28] E. Rojas-Vega and Z. Kemp, *An object-oriented data model for spatio-temporal data*, **Ninth Annual Symposium on Geographic Information Systems in Natural Resources Management. Symposium Proceedings**, Vol. 1, March 1995, pp. 399-406.

[29] D. Rosenzweig. *Distributed Computations: Evolving Algebra Approach*, **IFIP 13th World Computer Congress 1994**, Volume I: *Technology/Foundations*, In B. Pehrson and I. Simon (Eds.) Elsevier, Amsterdam, pp. 440-441.

[30] P. Sandiford, *IBEX: the electronic handshake greets electronic commerce*, **EDI Forum**, Vol: 8, Iss: 3, 1995, pp. 93-99.

[31] L. Segoufin, *Specifying active databases using evolving algebra*, **Mémoire de DEA, U. Paris XI, 1994.**

[32] D.W. Shipman, *The Functional Data Model and the Data Language DAPLEX*, **ACM Transactions on Database Systems**, Vol 6, 1981, pp. 140-173.

[33] Yelena Yesha, Nabil Adam, *Electronic Commerce: Current Research Issuses and Applications*, **Lecture Notes in Computer Science**, Eds. G. Goos, J. Hartmanis, J. van Leeuwen, Springer-Verlag, Vol. 1028, 1996.

[34] Yelena Yesha, Nabil Adam, *Electronic Commerce: An Overview*, **Electronic Commerce**, Eds. Nabil Adam, Yelena Yesha, **Lecture Notes in Computer Science**, Springer-Verlag, January, 1996.

[35] Yelena Yesha, Nabil Adam, Brad Fordham, *Digital Libraries and Electronic Commerce*, **Digital Libraries: Current Issues**, Eds. Nabil Adam, Bharat Bhargava and Yelena Yesha, Springer-Verlag, May 1995.