# Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems

EDWARD SCIORE
Boston College

MICHAEL SIEGEL
Massachusetts Institute of Technology

ARNON ROSENTHAL
The MITRE Corporation

Large organizations need to exchange information among many separately developed systems. In order for this exchange to be useful, the individual systems must agree on the meaning of their exchanged data. That is, the organization must ensure *semantic interoperability.* This paper provides a theory of *semantic values* as a unit of exchange that facilitates semantic interoperability between heterogeneous information systems. We show how semantic values can either be stored explicitly or be defined by *environments.* A system architecture is presented that allows autonomous components to share semantic values. The key component in this architecture is called the *context mediator*, whose job is to identify and construct the semantic values being sent, to determine when the exchange is meaningful, and to convert the semantic values to the form required by the receiver.

Our theory is then applied to the relational model. We provide an interpretaion of standard SQL queries in which context conversions and manipulations are transparent to the user. We also introduce an extension of SQL, called Context-SQL (C-SQL), in which the *context* of a semantic value can be explicitly accessed and updated. Finally, we describe the implementation of a prototype context mediator for a relational C-SQL system.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*query languages*; H.2.4 [**Database Management**]: Systems—*distributed systems*; H.2.5 [**Database Management**]: Heterogeneous Databases—*data translation*

E. Sciore was supported in part by ARPA grant F30602-93-C-0160. M. Siegel was supported in part by NSF grant IRI-90-2189, ARPA grant F30602-93-C-0160, and the International Financial Services Research Center at MIT. A. Rosenthal performed part of this work at ETH-Zurich.

Authors' addresses: E. Sciore, Computer Science Department, Carroll School of Management, Boston College, Chestnut Hill, MA 02167; M. Siegal, Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA 02139; A. Rosenthal, The MITRE Corporation, Burlington Road, Bedford, MA 01730.

General Terms: Design, Languages, Management

Additional Key Words and Phrases: Context, context mediator, conversion, data environments, heterogeneous database systems, semantic interoperability

## 1. INTRODUCTION

Large organizations need to exchange information among many separately developed systems. In order for this exchange to be useful, the individual systems must agree on the meaning of their exchanged data. That is, the organization must ensure *semantic interoperability.*

Obtaining and maintaining semantic interoperability are not easy. Over time, new data sources or applications may need to be added to an already heterogeneous mix. In addition to this, existing data sources may change the specifications of the data they provide, and applications may change the requirements for the data they receive. A resulting semantic conflict, if undetected, can produce disastrous results in even the simplest information system (such as source–receiver [Siegel and Madnick 1991]). However, the information needed to detect semantic conflicts is often buried deep in system manuals or users' minds, and the routines to resolve semantic conflicts, if existing, are buried in code. Imagine the organizational and technical problems that occur when a large organization with hundreds of applications and data sources determines that the meaning of some data has changed (e.g., unemployment numbers now include military personnel, stock prices are now reported as latest trade prices rather than as latest closing prices, or currencies are changed from French Francs to ECUs). How does it identify the impact of these changes on its applications? What does it take to adapt the application to the addition of new data sources? Currently, these problems are handled manually, resulting in the reduced availability and reliability of current information systems, increased requirements for training new workers, and a proliferation of anarchistic processes for determining when, where, and how information systems are affected by changes in the meaning of data.

In order to solve these problems, an information system must be able to ensure semantic interoperability. This is the goal of our research. We believe that, in order for this goal to be achieved, *context information* must be an active component of information systems. We define the context of a piece of data to be the metadata relating to its meaning, properties (such as its source, quality, and precision), and organization. For example, consider the trade price of a financial instrument, which might be reported as a number such as 101.25. The context of this number would be information such as its currency, source, precision, accuracy, scale factor, and status (e.g., is it the latest price or closing price?). A data value is exchanged from one system to another by converting it from its source context to its receiving context.

We have found numerous examples where explicit context specification would improve information systems in government, finance, and manufacturing. Context information can be quite varied, including information such as units specifications (e.g., currency, length), enumerated data tags (e.g., trade-price status), data or quality information (e.g., accuracy, source), data format (e.g., date format, scale factor), assumptions used in deriving or computing data (e.g., whether a write-off for a pension plan was excuded), methods (e.g., spline versus other interpolation strategies), or formula definition (e.g., Return on Equity defined to be Return/Equity, where Return is defined to be income and Equity is defined to be the number of shares times the share price on the close of the statement date) [McCarthy 1984; Wang and Madnick 1990]. When such context information is explicitly available, a database system can determine automatically whether data interchange is meaningful and identify effective means for converting data. The explicit context knowledge reduces errors and frees applications from being concerned with conversions. It also documents the differences among the environments of each system, making it possible to understand the impact of semantic changes to the data.

The foundation of our results is the concept of a *semantic value*, which is defined to be a piece of data together with its associated context. Section 2 presents a data-model-indpendent theory of semantic values. In it we show how *conversion functions* can be used to convert a semantic value from one context to another. We then use conversion to define what it means for two semantic values to be compared, as well as the meaning of some arithmetic operators on semantic values.

Conventional information system components (such as applications and database management systems) are designed for the exchange of simple values. Because semantic values, not simple values, more closely fulfill the requirements for information exchange, new system architectures must be developed. Whether this exchange involves one application calling another through a remote procedure call or an application requesting data from a data source, conventional systems must be enhanced with the ability to exchange semantic values. We call this new ability *context mediation*, and we call the system component responsible for context mediation the *context mediator*. In Section 3 we discuss this new architecture and the general requirements for context mediation. One important requirement is that a context mediator must be able to determine, given a simple value from the data source, the value's intended context in the data source as well as the value's expected context at the data receiver. This information is obtained by examining the *data environments* of the source and receiver. A data environment specifies the context of data values and may involve mappings, lookup tables, rules, predicates, or other knowledge representations.

Section 4 applies our theory to the relational model. We show how a semantic value can be modeled as several attribute values having a special relationship. For example, a relation may contain the two attributes *TradePrice* and *Currency*, with the value of *Currency* providing part of the context of *TradePrice*. We say that *Currency* is a *metaattribute* of *TradePrice*. This information is specified in the data environment of the database. For example, the data environment could specify that *Trade Price* has

*Currency* as one of its metaattributes and that the value of *Currency* in all tuples is '*USdollars*'. We show how a context mediator can use these environments to derive appropriate metaattribute values for database tuples. Because derived metaattribute values need not be stored, a traditional relational database can be treated as if it contains semantic values simply by associating with it a data environment that specifies values for all of its metaattributes.

We then examine how relations in our extended model are manipulated. In Section 5 we introduce an extension of SQL, called Context-SQL (C-SQL), that allows access, manipulation, and update capabilities for semantic values and their contexts. In C-SQL, context becomes a first-class construct. Thus, queries that before could only be determined by reviewing manuals or by asking fellow users can now be answered in C-SQL. Moreover, users see context information only when they wish to. A user posing a standard SQL query will not be aware of contexts and conversions, although these things may be involved in the evaluation of the query. For example, suppose a user poses an SQL query that compares two values of *Trade Price*. This query will be evaluated by converting these values to a common context and by comparing the resulting values, transparently to the user.

Section 5 also considers the effect of associating data environments with applications and user queries. We show how such data environments generalize the Application Semantic Views of Siegel and Madnick [1991], allowing users to access the data as if it actually had the context specified by the environment. For example, an application's data environment might specify that all currencies in a relation are to be in Yen. In this case, any SQL query on this relation could legitimately assume a single currency for all tuples and all attributes, even though the data in the underlying base relation involved multiple currencies. The conversion between currencies would be handled automatically and transparently by the context mediator. Update requests also take advantage of application data environments. For example, differences in the meaning of the data in the environment of an update and the environment of the data store can cause undetected data entry errors in conventional systems. These errors cannot happen if the update occurs through a semantic view. The system will ensure that the specified update values will be converted to the appropriate context before storing.

Section 6 considers implementation issues, describing a prototype context mediator for our extended relational model. This prototype is in the process of being developed at MIT.

Our work enforces an important trend toward decoupling of applications from other applications and data sources, thus simplifying the interface among system components. This decoupling is accomplished by defining semantic values as the unit of data interchange, allowing context to be specified both explicitly and implicitly, and providing an architecture that uses context mediators to interface with semantically heterogeneous systems. In Section 7 we summarize by examining the use of this approach in multidatabase systems and provide a description of future research plans.

## 2. SEMANTIC VALUES

In this section we provide the foundation for a theory of semantic values. Section 2.1 defines semantic values and motivates their use. Sections 2.2 and 2.3 discuss the idea of *conversions* and show how a semantic value can be converted to other semantic values. Section 2.4 shows how these conversions can be used to define *semantic comparability* and *semantic arithmetic*, and Section 2.5 examines some of the complexities in doing semantic comparisons in our current approach.

### 2.1 Properties and Contexts

A *simple value* is an instance of a *type*. The semantics of a simple value is determined solely by its type. That is, if *3* is of type *dollars* then it denotes 3 dollars and cannot be compared with instances of type *Yen* or *meters*.

Although typing provides some useful semantics to values [Siegel and Madnick 1989], it is inadequate when semantics have multiple dimensions. The main problem is that a different type is needed for every possible semantics; for example, an application manipulating currencies can have a different type for each kind of currency. Suppose, for example, that values can also be scaled; that is, *3* with a scaling factor of 1000 should denote 3000. We could encode this semantics by defining the types *thousands*, *millions*, etc., one type for each possible scaling factor. But, then, how do we combine currencies with scaling factors: What type do we assign to *3* so that it denotes 3000 Yen? Defining types such as *thousandYen* is both awkward and impractical.

In order to increase the semantics of values, we turn to LISP's idea of property lists. In LISP at atom has an associated list of properties. For example, a LISP atom might have the properties *Currency* and *ScaleFactor*; if the value of the atom is *3*, the value of its *Currency* property is '*Yen*', and the value of its *ScaleFactor* property is *1000*, then the atom denotes 3000 Yen.

Formally, we define a *semantic value* to be the association of a *context* with a simple value. We define a context to be set; each element of the set is an assignment of a semantic value to a *property*. Note that this definition is recursive. That is, the value of a property can have a nonempty context. Simple values are defined to be equivalent to semantic values having an empty context.

We write semantic values by placing the context of a simple value next to it, using parentheses instead of set brackets. For example, the following semantic value might appear in a stock market application:

$$\begin{aligned}&1.25(\mathit{Periodicity} = \text{`}\mathit{quarterly}\text{'}(\mathit{FirstIssueDate} = \text{`}\mathit{Jan.\ 15}\text{'}),\\ &\qquad \mathit{Currency} = \text{`}\mathit{US\ dollars}\text{'}).\end{aligned}$$

Here, the value *1.25* has two properties: *Periodicity* and *Currency*. The value of the former property is also a semantic value having the property *First-*

*IssueDate*. The semantics of *1.25* can thus be interpreted as a quarterly dividend of 1.25 U.S. dollars with a beginning cycle of January 15.

## 2.2 Conversion Functions

Attaching a context to a simple value helps to specify the value's semantics more accurately. One consequence of making context explicit is that two syntactically different semantic values can have the same meaning, for example, *4(LengthUnit ='feet')* and *48(LengthUnit ='inches')*. Intuitively, these semantic values are equivalent because there exist *conversions* between them. In this subsection we discuss conversions between semantic values having a single property. The next subsection extends these results to arbitrary semantic values.

Let *P* be a property. A *conversion function* for *P* is a function that converts a simple value from one value of *P* to another. For example, if *cvtLengthUnit* is a conversion function for *LengthUnit*, then *cvtLengthUnit(4,'feet','inches')* returns the simple value *48*. A conversion function may be implemented in any programming language and may involve table lookup (e.g., for currency conversion), consulting on-line data sources (e.g., for timely currency conversion), or logical rules.

It is possible for a property to have more than one conversion function defined for it. For example, consider the property *PriceStatus*, whose possible values include *'latestTradePrice'*, *'latestNominalPrice'*, and *'latestClosingPrice'*. There is no universal interpretation of these values. One application might have a complex formula that estimates the latest closing price given a latest trade price (e.g., as a function of the S & P 500 index); another application might ignore the differences between the two *PriceStatus* values, so that a semantic value such as *3(PriceStatus ='latestTradePrice')* can be converted to *3(PriceStatus ='latestClosing Price')* and back again; yet another application might not allow any conversions, treating the different kinds of price status as being incomparable. These different choices are all reasonable and simply depend on how an application intends to use its data.

Conversion functions can be defined by the system or by any of the databases or applications that are part of the system. The conversion functions are stored in one or more *conversion libraries*. In Section 3 we discuss the general case, in which the context mediator must determine the appropriate conversion function to use in a given situation. However, for simplicity we assume in this paper that all properties *P* have a single conversion function, which is called *cvtP*.

A conversion function may have characteristics that are useful for semantic comparison, as we shall see in Section 2.4. In particular, a conversion function may be *total*, *lossless*, or *orderpreserving*. A *total* conversion function is one that is defined for all arguments. The function *cvtLengthUnit* is an example of a total conversion function, because it is possible to convert any value from any unit to any other unit. Not all conversion functions are total; for example, it might not be possible to convert from one currency to another. For another example of a nontotal conversion function, consider semantic values denoting locations of companies. Assume that locations are represented as character strings, such as '*Boston*', '*Paris*', '*Alaska*', or '*Spain*'. This set of location values can be thought of as forming a hierarchy; for example, '*Paris*' is less general than '*France*'. We introduce the property *LocationGranularity* to encode the granularity of the location, whose conversion function *cvtLocationGranularity* converts strings from one granularity to another. Thus, '*France*' = *cvtLocationGranularity*('*Paris*', '*city*', '*country*'). This conversion function is not total—because it is not possible to convert from a coarse granularity to a finer one.

A *lossless* function is one for which it makes no difference whether a value is converted from one property value to another directly or in a sequence of steps. Formally, *cvtP* is a lossless conversion function if $cvtP(a, p_1, p_1) = a$ and $cvtP(a, p_1, p_3) = cvtP(cvtP(a, p_1, p_2), p_2, p_3)$ for any values of $a$, $p_1$, $p_2$, and $p_3$. For example, *cvtLengthUnit* is a lossless conversion function, because by starting with a value in a particular unit and performing an arbitrary number of conversions on it, the resulting value is always the same as what would result from converting it directly to the final unit. Note that a special case of this definition occurs when $p_1 = p_3$; in this case, the definition asserts that converting from one property value to another and back again results in what you started with.

Examples of lossy (i.e., nonlossless) conversion functions include those that convert between different geometric formats (line segment representations versus spline representations) or different discrete representations of continuous data, as well as those that perform lossy data compression. For another example of a lossy conversion function, consider semantic values denoting text strings containing embedded formatting commands. A semantic value such as *s*(*FormatType* ='*LaTex*') indicates that string *s* contains embedded LaTeX commands. Assume that the function call *cvtFormatType*(*s*,'*LaTex*', '*unformatted*') returns a string *s'* in which the embedded commands are removed, whereas the call *cvtFormatType*(*s'*, '*unformatted*', '*LaTex*') returns a string in which a default set of formatting commands is added to *s'*. Clearly, this function is not lossless, because if you take a LaTeX string, and convert it to unformatted and then back to LaTeX, you may get something different from what you started with.

We say that a conversion function for property $P$ is *order preserving* if, for any values $p$ and $p'$ of $P$, $a_1 < a_2$ implies that $cvtP(a_1, p, p') < cvtP(a_2, p, p')$. Intuitively, a converson function is order preserving if the simple values associated with two semantic values do not change their order when they are converted to another context. Again, the function *cvtLengthUnit* is order preserving: The semantic value for 3 inches will always be less than the one for 4 inches, no matter what units they are converted to. For an example of a non-order-preserving property, consider the property *CodeType*, which documents whether an integer corresponds to a character in either the ASCII or EBCDIC code. For example, *48*(*CodeType* ='*ascii*') and *240*(*CodeType* ='*ebcdic*') both correspond to the character '*0*'. Similarly, the character '*A*' is represented by the semantic values *65*(*CodeType* ='*ascii*') and *193*(*CodeType* ='*ebcdic*'). Thus, the conversion function *cvtCodeType* is not order preserving, because *cvtCodeType(48, 'ascii', 'ebcdic')* > *cvtCodeType(65, 'ascii', 'ebcdic')*, even though 48 < 65.

Each conversion function has an associated cost function, which estimates the cost of a given conversion. For example, conversions involving consulting on-line sources will have a higher cost than conversions involving a simple arithmetic calculation. Knowledge about conversion costs is important for query processing, as we shall see in Section 3.2. Conversions that are not possible are assigned an arbitrarily high cost. Additionally, conversions that lose information can be discouraged through a higher cost.

One final issue to discuss is the treatment of conversions between property values that have their own context. For example, Section 2.1 introduces the property *Periodicity*, the values for which contained bindings for the property *FirstIssueDate*. A sample call to the conversion function would thus be

$$\begin{gathered} \mathit{cvtPeriodicity}(6, \text{`}\mathit{quarterly}\text{'}(\mathit{FirstIssueDate} = \text{`}\mathit{Jan.\ 15}\text{'}), \\ \text{`}\mathit{annually}\text{'}(\mathit{FirstIssueDate} = \text{`}\mathit{June\ 1}\text{'})). \end{gathered}$$

Note that the interpretation of the property *FirstIssueDate* is hidden inside the conversion function *cvtPeriodicity*. This conversion function has several possible options: It can ignore the context of its arguments; it can require that the contexts be identical; it can call the conversion function *cvtFirstIssueDate*; and it can perform internal calculations based on the context values. Although our theory allows for property values to contain other properties, the best way to use such properties is unclear and is the subject of future research.

## 2.3 Converting Arbitrary Contexts

In this subsection we discuss conversions among semantic values having more than one property. Let $v$ be an arbitrary semantic value, and let $C$ be a context containing values for a subset of the properties in the context of $v$. Then we define the function $\mathit{cvtVal}(v, C)$ to return the semantic value that results from converting $v$ to context $C$. For example,

$$\begin{gathered} \mathit{cvtVal}(40(\mathit{LengthUnit} = \text{`}\mathit{feet}\text{'}, \mathit{scaleFactor} = 1), \\ \{\mathit{LengthUnit} = \text{`}\mathit{inches}\text{'}, \mathit{ScaleFactor} = 10\}) \end{gathered}$$

returns the semantic value $48(\mathit{LengthUnit} = \text{`}\mathit{inches}\text{'}, \mathit{ScaleFactor} = 10)$.

The function call $\mathit{cvtVal}(v, C)$ returns a semantic value. The context of this return value is $C$, and its simple value is obtained by composing conversion functions. In particular, let $C$ be the context $\{P_1 = p_1, \ldots, P_n = p_n\}$, let $p_i'$ be the value of $P_i$ in the context of $v$, and let $a$ be the simple value of $v$. Then the simple value of the semantic value returned by the function call is

$$\mathit{cvtP}_n(\ldots, \mathit{cvtP}_2(\mathit{cvtP}_1(a, p_1', p_1), p_2', p_2), \ldots, p_n', p_n).$$

Note that any property values appearing in the context of $v$ but not in $C$ are ignored in the conversion.

In the above example involving *cvtVal*, the order in which the conversion functions *cvtLengthUnit* and *cvtScaleFactor* are composed does not matter;

both orderings produce the same value. We say that these conversion functions *commute*. Conversion functions may not commute, in general. For example, a function that performs units conversion will not commute with a function that converts between encrypted and unencrypted values. In such cases, we need a way to specify the order in which the conversion functions are to be composed. We therefore require that a *priority* be specified with each conversion function. The order in which conversion functions are composed in the above definition is determined by their priority; in particular, functions having a higher priority are evaluated first. Conversion functions that commute can be given equal priority, in which case the system is free to choose any ordering.

### 2.4 Semantic Comparability

We now turn to the issue of what it means to compare two semantic values. In general, the result of a comparison is a relative thing. For example, consider the semantic values *4(Currency ='USdollars')* and *300(Currency ='Pesetas')*, and suppose that currency conversion is not lossless. In particular, suppose that *cvtCurrency(4, 'USdollars', 'Pesetas') = 300*, but *cvtCurrency(300,'Pesetas', 'USdollars') = 3*. Then the above two semantic values should be considered equal if we are interested in their worth in Pesetas, but not if we are interested in their worth in U.S. dollars. For another example, consider the semantic values *'Paris'(LocationGranularity ='city')* and *'France'(LocationGranularity ='country')*. Here, the locations should be considered equivalent if we are interested in whether they denote the same country, but should be inequivalent if we are interested in whether they denote the same city. The intuition behind these examples leads to the following definition:

*Definition.* Let $v_1$ and $v_2$ be two semantic values, and let $C$ be a context. Suppose that $cvtVal(v_1, C) = a_1(C)$ and $cvtVal(v_2, C) = a_2(C)$. Let $\theta$ be a comparison operator, such as $=$ , $\neq$ , $<$ , and so on. Then we say that $v_1 \ \theta \ v_2$ *with respect to C* if $a_1 \ \theta \ a_2$.

That is, two semantic values must be compared with respect to a context. This context is called the *target context* of the comparison. The value of the comparison is defined by converting both semantic values to the target conext and by comparing the associated simple values of the results. We call this kind of comparison *relative comparison*, because the truth value of the comparison depends on the target context.

The target context may be different from the contexts of either of the compared values. For example, *'Paris'(LocationGranularity ='city')* $=$ *'Nice'(LocationGranularity ='city')* with respect to the context *{LocationGranularity ='country'}*. The target context also may have different properties than either of the compared values. The only restriction is that the properties of the target context be contained in the intersection of the property sets of the compared values; this restriction arises from the definition of *cvtVal*. In the case that the target context is empty, no conversion need be done, and semantic equality reduces to equality on the simple values. For example, let $v_1$ be *4(LengthUnit ='feet', ScaleFactor = 10)*, and let $v_2$ be *40(LengthUnit ='meters', ScaleFactor = 1)*. Then, $v_1 = v_2$ with respect to the context *{ScaleFactor = 17}* (or any other context involving just *ScaleFactor*) because the differences in units are ignored; similarly, $v_1 < v_2$ with respect to the empty context because all properties are ignored.

Arithmetic operators such as addition and subtraction can also be performed on semantic values. In general, these operations must be evaluated in a context, and the result of an operation will be a semantic value having the target context. For example, the result returned by the operation *3(Currency ='USdollars')* + *300(Currency ='Peseas')* depends on the currency we wish to see. Formally, suppose that $cvtVal(v_1, C) = a_1(C)$ and $cvtVal(v_2, C) = a_2(C)$ for semantic values $v_1$ and $v_2$, respectively. Then $v_1 + v_2 = a_3(C)$ with respect to $C$ if $a_1 + a_2 = a_3$.

When two semantic values are being compared, it is often the case that only the properties belonging to the target context need to be specified; all choices of values for these properties result in the same answer. For example, let $v_1$ = *6(LengthUnit ='inches', ScaleFactor = 10)* and $v_2$ = *5(LengthUnit ='feet', ScaleFactor = 1)*. Then $v_1 = v_2$ not only with respect to the context *{LengthUnit ='meters', ScaleFactor = 35}*, but also with respect to any context involving the properties *LengthUnit* and *ScaleFactor*.

Formally, let $v_1$ and $v_2$ be arbitrary semantic values, and let $S = \{P_1, \ldots, P_n\}$ be a set of properties. If $v_1 = v_2$ for any context involving the properties in $S$, then we say that they are *absolutely equal* with respect to $S$. Absolute inequality, greater than, less than, and so on are defined similarly.

Let $S = \{P_1, \ldots, P_n\}$ be a set of properties. We can show that, if every $P_i$ is total and lossless, then any semantic comparison with respect to $S$ involving $=$ or $\neq$ will be absolute. We outline the proof by considering the special case of equality where $S$ contains a single property $P$; the proofs of the general cases for equality and inequality are similar. So suppose that $a_1(P = p_1) = a_2(P = p_2)$ with respect to $\{P = p\}$. Then, by definition, $cvtP(a_1, p_1, p) = cvtP(a_2, p_2, p)$. Let $a$ be the value returned by the calls to $cvtP$. Then the fact that $cvtP$ is total implies that $cvtP(a, p, p')$ is defined for any $p'$. The fact that $cvtP$ is lossless implies that $cvtP(a, p, p') = cvtP(a_1, p_1, p') = cvtP(a_2, p_2, p')$. Thus, $a_1(P = p_1) = a_2(P = p_2)$ with respect to $\{P = p'\}$ as well.

In order for two semantic values to be absolutely comparable via greater than or less than, the properties in the target context must also be order preserving; this restriction ensures that, when the semantic values are converted to $p'$, they do not change their relative order. The proof of this claim is straightforward and is similar to the one above for equality. As an example, *cvtScaleFactor* is order preserving, and thus, *6(ScaleFactor = 10)* < *4(ScaleFactor = 100)* absolutely with respect to *{ScaleFactor}*. On the other hand, we have seen that the function *cvtCodeType* is not order preserving. This lack of order preservation makes any definition of absolute less than meaningless in this domain. For example, *65(CodeType ='ascii')* >

*240(CodeType ='ebcdic')* with respect to *{CodeType ='ebcdic'}*, but the opposite is true if the comparison is with respect to *{CodeType ='ascii'}*.

### 2.5 Resolvable and Nonresolvable Properties

In the previous subsection, we assumed that two semantic values could be compared by converting them to the same context and then by comparing the simple values. This assumption, however, is not always true. For example, consider the property *Precision*, which encodes information about the accuracy of a number. That is, the semantic value $v_1$ = *3.01(Precision = 0.1)* denotes a number whose value is known to be between 3.11 and 2.91. Now consider the value $v_2$ = *2.99(Precision = 0.1)*. Our definitions imply that $v_1 > v_2$, but this result is not intuitive. In fact, an application may reasonably went to treat these numbers as equal. Consequently, we distinguish between two kinds of properties: *resolvable* and *nonresolvable* properties. Formally, let $P$ be a property, let $v_1 = a_1(P = p)$ and $v_2 = a_2(P = p)$ be two semantic values having context $\{P = p\}$, and let $\theta$ be a comparison operator. Then we say that $P$ is a *resolvable* property if, for any values of $a_1$, $a_2$, $p$, and $\theta$, $a_1 \, \theta \, a_2$ implies that $v_1 \, \theta \, v_2$ with respect to context $\{P = p\}$.

Intuitively, a resolvable property is one for which semantic comparison can always be reduced to the comparison of simple values. If a property is not resolvable, then the only way to perform semantic comparison is for the application to supply a specific comparison function for each comparison operator. For example, the only way to test whether *3.01(Precision = 0.1)* > *2.99(Precision = 0.4)* is to feed these semantic values directly to the appropriate comparison function for *Precision*; conversion in this case is not sufficient.

Data quality metrics [Wang 1992] such as accuracy and precision are good examples of nonresolvable properties. Each application is likely to have a specific approach to semantic comparison and semantic arithmetic for such properties. The theory developed in the previous subsections applies only to resolvable properties. All examples encountered in previous subsections are resolvable, and we shall consider only resolvable properties in the rest of this paper. It is an interesting open problem to extend our results to nonresolvable properties.

## 3. AN ARCHITECTURE FOR EXCHANGING SEMANTIC VALUES

Conventional information system components (e.g., applications and database management systems) are designed for the exchange of simple values. However, this form of information exchange does not extend to real-world systems where the meaning of exchanged values can change. Semantic values, not simple values, more closely fulfill the requirements for information exchange. However, current applications and data sources are not equipped to send or receive data as semantic values, as they cannot evaluate properties, determine semantic comparability, select target contexts, and resolve conflicts.

A new system architecture is needed to facilitate the exchange of semantic values among its component information systems. In this section we propose such a system architecture, which provides a general solution to semantic interoperability and accommodates existing component systems and data models. Sections 4–6 then describe how our system architecture can be applied to the relational model.

### 3.1 The Architecture's Components

Our proposed architecture contains five kinds of components: information systems, data environments, context mediators, conversion libraries, and shared ontologies. Figure 1 depicts this architecture for a source–receiver model [Siegel and Madnick 1991], in which there are two component information systems (the data source and the data receiver) and a single context mediator. A discussion of how this architecture extends to a generalized client/server model appears in Siegel et al. [1993].

The central component of this architecture is the *context mediator*. The context mediator is the agent that directs the exchange of values from one component information system to another, and provides services such as intersystem attribute mapping, property evaluation, and conversion. All data exchange goes through the context mediator. A context mediator is designed to function between specific component system interfaces (e.g., SQL or remote procedure calls) and is modified only when the interface is modified. The context mediator is an example of the mediator concept of Wiederhold [1992]. This approach has two advantages: First, it is nonintrusive, in the sense that there is no constraint on the data models used by the component information systems. Second, it limits the number of interfaces required of each system component. The workings of the context mediator are described in more detail in Section 3.2.

Each information system component may have an associated *data environment*. A data environment contains two parts: Its *semantic-value schema* specifies attributes and their properties, and its *semantic-value specification* specifies values for some or all of these properties. The context mediator uses the data environments to determine whether a requested data exchange is possible and, if so, what conversions are necessary. Details of this process are given in Section 3.3.

The *shared-ontology* component specifies *terminology mappings*. These mappings describe naming equivalences among the component information system, so that references to attributes (e.g., *Exchange* or *CompanyName*), properties (e.g., *Currency*), and their values (e.g., *'USdollar'*) in one information system can be translated to the equivalent names in another. The development and implementation of an ontology are complex problems and are part of a basic goal in enterprise modeling at both a technical [Collet et al. 1991; Neches et al. 1991] and organizational level.

The final kind of component is the *conversion library*. A conversion library contains conversion functions; there is a global converson library, as well as a local conversion library for each information system component. A property may have conversion functions in several libraries, and a library may have multiple conversion functions for a given property. For example, consider the property *Currency*. The global conversion library may contain a large number

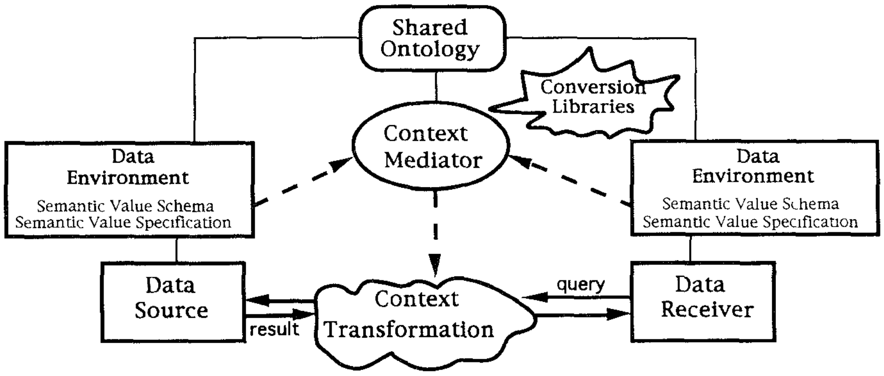

Fig. 1. Architecture for semantic interoperability using semantic values.

of conversion functions for this property, according to different dates and locations.

Because our proposed system architecture is modular, it is easily extensible. New information system components can be added, with their associated data environments. An existing data environment may change to incorporate the addition of new properties or changes in the values assigned to existing properties. Conversion functions can be added or removed from conversion libraries. Additional context mediators will be needed only when components with different interfaces are added or an interface between existing components is modified.

## 3.2 Context Mediation

We model the exchange of data values as follows: The receiver information system sends a query to the context mediator, requesting some semantic values from the source information system and provding target context for the result. The context mediator then sends this query (possibly transformed) to the data source and receives the resulting values. The context mediator then converts these values to the given target context and sends the result to the receiver. This process implies several requirements for the context mediator.

First, the context mediator must be able to analyze the query and to determine the most appropriate strategy for performing conversions. This strategy may involve composing intermedate queries and creating temporary relations and views. For example, suppose the receiver requests the names of those companies whose trade price is greater than 50 U.S. dollars. If the context mediator can determine that currencies in the data source are always in Yen,[1] then it may choose to transform the query into one that mentions the amount of Yen equivalent to 50 U.S. dollars. On the other hand, if such a

[1] Such a determination can be made by examining the data environment of the source; see Section 3.3.

determination is not possible then the context mediator might choose to convert all of the source values to U.S. dollars before applying the query. That is, the context mediator could choose the following strategy: First, it would send a query to the source, requesting all companies and their trade prices; next, it would convert the retrieved trade prices to U.S. dollars and save the resulting values temporarily at the source; finally, it would request from this temporary relation the names of the companies having trade-price values of more than 50, sending the results to the receiver. This latter strategy is adopted by our prototype context mediator of Section 6.

This ability to develop a strategy for performing conversions is also needed to support semantic comparison. Here, the receiver is requesting the result of comparing two source values. In general, the context mediator will have different possible strategies for choosing the target contexts of each semantic comparison. At one extreme, it may choose to perform comparisons in the context of the source as much as possible, converting only the final return values to the target context of the receiver. At the other extreme, it could choose to convert all values in the source to the target context, evaluating the comparisons at that point. The extent to which it chooses one extreme or the other is a query optimization issue.

Second, the context mediator must be able to compare the context of the resulting semantic values with the target context, in order to ensure that the exchange is meaningful. In particular, the context mediator must verify that the properties of the target context are a subset of the properties of each semantic value's context, so that *cvtVal* will be used properly (recall Section 2.3). In performing this comparison, the context mediator may need to access the terminology mappings in the shared-ontology component. For example, suppose that the data source returns the semantic value *30(Currency ='Yen', PriceStatus ='latestTradePrice')* and that the target context of the receiver is *{TradeCurrency ='USdollars'}*. The context mediator would use the terminology mappings to note that *Currency* and *TradeCurrency* are different names for the same property. It then would note that the target context is a subset of the source context, and therefore, conversion is meaningful. On the other hand, suppose that the source returns the semantic value *30(Currency ='Yen')* and that the target context is *{TradeCurrency ='USdollars', Price Status ='latestTradePrice'}*. Then for context mediator would determine that the conversion is not possible because there is no value for *PriceStatus* in the source context, and therefore, the request for data fails.

Third, the context mediator must be able to locate the appropriate conversion functions. A conversion function is located by first searching the conversion library of the receiver for the desired conversion functions; the global conversion library is searched to find those conversion functions not in the local library. If a library has more than one conversion function for a property, the context mediator must choose one of them. This choice can be based on system defaults (such as by choosing the most recent function), based on specifications appearing in the receiver's data environment, or made by interactively requesting information at run time. For example, in a query involving currency conversion the receiver may indicate the effective time, date, and location for the conversion in its data environment.

Fourth, the context mediator must perform the appropriate conversion on each resulting semantic value $v$. In order to determine the best conversion plan, the mediator must examine the properties, cost, and characteristics (i.e., lossless, total, or order preserving) of the relevant conversion functions, and thereby determine the best evaluation sequence for the expression $cvtVal(v, C)$. In more complex examples, the mediator must distinguish between resolvable and nonresolvable properties, and evaluate context-based arithmetic and comparison expressions.

The fifth and final requirement is for the context mediator to provide the results to the receiver. In the simplest case, the context mediator will pass the values returned by *cvtVal* to the receiver. However, if some conversions are not possible, the context mediator must decide whether to abort the query and return an appropriate error message, or to pass a partial result to the receiver with an explanation of the semantic conflicts [Siegel and Madnick 1991]. If a partial result is returned, the context mediator must provide a way to inform the receiver that these results are due to a modification that limited the scope of the query.

### 3.3 Data Environments

In the previous subsection, we have assumed that the context mediator interfaces with the source and receiver by means of semantic values. However, this is not necessarily true; existing systems are designed to manipulate simple values only. Consequently, we introduce *data environments* as a way for the context mediator to turn the simple value provided by the source into a semantic value that can be exchanged.

Formally, a data environment has two parts: a *semantic-value schema* and a *semantic-value specification*. The semantic-value schema declares the properties associated with each semantic value, and the semantic-value specification specifies values for some or all of these properties for a given simple value. Semantic-value specifications can be declared in many ways:

—by rules [Siegel and Madnick 1991];
—by predicates in a logic [Collet et al. 1991; Shen et al. 1991];
—by functional expressions;
—by tables (including virtual tables); and
—by tagged attribute properties (such as *source*, *quality*, and *security*) [Wang and Madnick 1990].

In Section 4.2 we give examples of data environment specifications using rules and predicates in the relational model.

Data environments are also associated with the receiver. If the receiver does not explicitly specify a target context, then the context mediator can use the receiver's data environment to determine the target context of the exchange. If the receiver does not handle semantic values, then the context mediator converts the source value to the target context and then passes the simple value only to the receiver. Examples of this process appear in Sections 5 and 6.

## 4. RELATIONAL DATABASES AS INFORMATION SOURCES

In our proposed system architecture, each information source has an associated data environment that assists the context mediator in its exchange of semantic values. In this section we consider the case where the information source is a relational database. We discuss the meaning of semantic values in the relational model, the declaration of a data environment's semantic-value schema and semantic-value specification, and the role of context mediation in query processing.

### 4.1 Semantic-Value Schemata

In the relational model, a *relation schema* is defined to be a set of *attributes*, each of which has a specified *type*. A *relation* is a set of *tuples*; a tuple contains a value for each attribute in the relation schema. Attribute values in the relational model are simple values. Conceptually, however, a database needs to supply semantic values to the context mediator. We resolve this problem by modeling a semantic value as a subtuple of simple values, with each property of the semantic value corresponding to an attribute. Attributes corresponding to properties are called *metaattributes*; the other attributes are called *base attributes*. Note that both base attributes and metaattributes may have associated metaattributes.

The semantic-value schema is the place where attributes and their metaattributes are declared. Figure 2 gives a semantic-value schema for a financial database containing the two relations *TRADES* and *FINANCES*. The syntax of the semantic-value schema declaration is an extension of the SQL syntax for declaring relation schemata; the difference is that the semantic-value schema specifies the association between each attribute and its metaattributes. This association is achieved by placing the declaration of a metaattribute after its associated attribute within nested parentheses. In Figure 2 the *TRADES* relation has the five base attributes *CompanyName*, *InstrumentType*, *Exchange*, *TradePrice*, and *Volume*; each tuple in this relation records the trading of a financial instrument (such as a company's stock) on an exchange. The *FINANCES* relation has the five base attributes *CompanyName*, *Location*, *Revenues*, *Expenses*, and *Dividend*; each tuple in this relation records some of the financial information about a company. Sample domain values for some of these base attributes and metaattributes appear in Figure 3.

Several attributes may define a metaattribute having the same name. For example, a metaattribute named *Currency* is defined in four places in Figure 2. Consequently, metaattributes are identified by qualifying their name with the name of their associated attribute. For example, the metaattributes of *TradePrice* are *TradePrice.PriceStatus* and *TradePrice.Currency*. The

```
create table TRADES
    (CompanyName char(50),
     InstrumentType char(10),
     Exchange char(20),
     TradePrice float4
         (PriceStatus char(20),
          Currency char(15)),
     Volume int
         (Scalefactor int,
          VolumeStatus char(15)))

create table FINANCES
    (CompanyName char(50),
     Location char(40)
         (LocationGranularity char(15)),
     Revenues float4
         (Scalefactor int,
          Currency char(15)),
     Expenses float4
         (Scalefactor int,
          Currency char(15)),
     Dividend float4
         (Periodicity char(10)
              (FirstIssueDate date),
          Currency char(15)))
```

Fig. 2. Semantic-value schema.

metaattribue of *Dividend.Periodicity* is called *Dividend.Periodicity.FirstIssueDate*.

Metaattributes, because they correspond to properties, may have conversion functions associated with them. These conversion functions are stored in conversion libraries, as discussed in Section 3.1. As before, we assume that the conversion function for metaattribute $P$ is called *cvtP*. Additionally we assume for the rest of this paper that all conversion functions on ordered properties are order preserving, and that the only conversion function that is not lossless and total is *cvtLocationGranularity*, which behaves as described in Section 2.2.

### 4.2 Semantic-Value Specifications

The metaattributes defined for a relation are real attributes, in the sense that their values can be accessed by queries. However, this does not mean that metaattributes must be explicitly stored; indeed, there are many times when this is undesirable. Metaattribute values in a relation stored; indeed, there are many times when this is undesirable. Metaattribute values in a relation or database often have a regular, well-defined pattern. This regularity might be a consequence of behavior in the real world (e.g., "All U.S. companies report earnings in U.S. dollars"), business rules (e.g., "Dividends are always issued quarterly"), or characteristics of the database is chosen by its DBA (e.g., "All *Volume* values are stored with a scale factor of 1000"). A

```
domain(InstrumentType) = {'equity', 'warrant', 'future'}
domain(Exchange) = {'nyse', 'madrid'}
domain(TradePrice.Currency) = {'USdollars', 'FrenchFrancs', 'Pesetas', 'Yen'}
domain(TradePrice.PriceStatus) = {'latestTradePrice', 'latestNominalPrice', 'latestClosingPrice'}
domain(Volume.VolumeStatus) = {'latestVolume', 'closingVolume'}
```

Fig. 3. Examples of some attribute and metattribute domains.

```
create scene for TRADES by rules
   if InstrumentType = 'equity' and Exchange = 'madrid'
      then TradePrice.PriceStatus = 'latestClosingPrice' and
           TradePrice.Currency = 'Pesetas';
   if InstrumentType = 'equity' and Exchange = 'nyse'
      then TradePrice.PriceStatus = 'latestTradePrice' and
           TradePrice.Currency = 'USdollars';
   if InstrumentType = 'future'
      then TradePrice.PriceStatus = 'latestClosingPrice' and
           Currency = 'USdollars';
   if InstrumentType = 'equity'
      then Volume.VolumeStatus = 'latestVolume' and
           Volume.Scalefactor = 1;
   if InstrumentType = 'future'
      then Volume.VolumeStatus = 'closingVolume' and
           Volume.Scalefactor = 1000;

create scene for TRADES, FINANCES by predicate
   Currency = 'USdollars' and ScaleFactor = 1
```

Fig. 4. Semantic-value specification with two scenes.

database is often subject to standards and regulations that restrict the possible metaattribute values. It is therefore desirable for the value of such metaattributes to be computed instead of stored. Such computation is specified in the semantic-value specification of the database.

In the relational model, a semantic-value specification consists of one or more parts, with each part applying to some set of relations. We call each part a *scene*. Figure 4 presents syntax for a semantic-value specification containing two parts: a scene for *TRADES* and one for {*TRADES*, *FINANCES*}. The former scene is defined by rules, whereas the latter scene is defined by a predicate. Note that metaattributes need not be prefixed by attribute names. The meaning of the term *"Currency ='USdollars'"* in the latter scene of Figure 4 is that the value for the metaattribute *Currency*, in every appropriate attribute of all relations in the group, is *USdollars*.

A metaattribute whose value is defined by a semantic-value specification is called a *derived metaattribute*. If more than one scene defines values for the same metaattribute, then a conflict-resolution heuristic similar to standard single inheritance is used. In particular, we say that scene $S_1$ has higher priority than scene $S_2$ if the set of associated relations of $S_1$ forms a subset of the associated relations of $S_2$. If there is a highest-priority scene that assigns a value to a given metaattribute, then that scene is used; if there is no such scene, then an error occurs. For example, suppose that tuple $t$ has just been inserted into relation *TRADES*. If the scene for *TRADES* can be used to determine a value of the metaattribute *ScaleFactor*, then that value is used. Otherwise, the scene for {*TRADES*, *FINANCES*} is used, and the value of *ScaleFactor* is set to 1.

The value of a derived metaattribute is fixed by the semantic-value specification and cannot be overridden. Consequently, if a tuple is inserted into a relation, then its derived metaattributes must have the values indicated by the semantic-value specification. If a user specifies a metaattribute value that is different from what it should be, then the tuple will have to be converted appropriately before it is stored. The details of this process appear in Section 5.2.

Every information system makes some tacit assumptions about the data it contains. When a database is used in a wider setting than originally anticipated, its assumptions need to be made more explicit. This is one of the fundamental requirements for interoperability in heterogeneous database systems and can be achieved by means of derived metaattributes in semantic-value specifications. In particular, we note that derived metaattributes need not be physically stored in a relation because their values for a given tuple can always be calculated. A semantic-value specification in which all metaattributes are derived was called a *Database Metadata Dictionary* (*DMD*) in Siegel and Madnick 1991. Consequently, an existing relation need not be changed in order to include metaattributes; all that is necessary is for a scene to be declared for it.

The presence of derived metaattributes also allows the context mediator to reason about the contents of these metaattributes without accessing the database. This ability can be used for query optimization, as we shall see in Section 6.

## 5. RELATIONAL APPLICATIONS AS INFORMATION RECEIVERS

In the previous section, we have considered relational information sources. We now consider information receivers in the relational model.

An information receiver can be an application program or an interactive user. We assume that a receiver accesses an information source by formulating a query in some data manipulation language. In this section we use C-SQL as that language. C-SQL is our extension of SQL, which provides the capability for queries to take greater advantage of the presence of metaattributes. The semantics of C-SQL is illustrated by means of numerous example queries. All of these queries refer to the source database of Section 4, which has the semantic-value schema shown in Figure 2 and the semantic-value specification shown in Figure 4. We begin this section by assuming that the receiver issuing these queries has the same data environment as the source database. In Section 5.4 we consider the case when the receiver has a different semantic-value specification from the source. The results of this section extend the preliminary ideas of Sciore et al. [1992].

## 5.1 Queries

We begin by considering the meaning of a standard SQL query in the presence of data environments. For example, suppose that the receiver submits the following query:

> **select** *t1.CompanyName*, *t1.Location*
> **from** *FINANCES t1 t2*
> **where** *t2.CompanyName* =*'IBM'* **and** *t1.Expenses* > *t2.Expenses*

Intuitively, this query requests the source to retrieve the name and location of those companies having expenses greater than those of IBM. However, the use of semantic values as the unit of exchange affects the meaning of the query in two ways: First, the values appearing in the output tuples are composed of *semantic* values. That is, not only is the location of the company retrieved, but its granularity is as well. Second, all comparisons in the **where** clause are *semantic* comparisons. Semantic comparison requires a target context. Because SQL has no means for specifying such a context, we adopt conventions to infer the context from the query. In particular, we use the intersection of the metaattributes of the values being compared to determine the metaattributes of the target context. If the comparison is absolute, then this is all that is needed; the system may choose any values for these metaattributes without affecting the truth value of the comparison. If the comparison is relative, then values for these metaattributes must be the same; if the values are not the same, then there is no obvious target context for the system to choose, and the query is aborted. In Section 5.4 we show how this apparent difficulty can be avoided by an appropriate use of semantic-value specifications in queries. Until then, we shall consider only absolute comparisons in our examples.

There are two comparisons in the above query. The first comparison is *t2.CompanyName* =*'IBM'*. Inasmuch as *CompanyName* has no metaattributes, the target context is empty, and the semantic equality reduces to simple equality. The second comparison is *t1.Expenses* > *t2.Expenses*. The attribute *Expenses* has the two metaattributes *Currency* and *ScaleFactor*, both of which have lossless, total, and order-preserving conversion functions; thus, absolute semantic comparisons are possible. Each semantic value for *t1.Expenses* will be compared with each semantic value for *t2.Expenses* using semantic greater than.

Note the advantage that semantic comparison brings to this query: Even though the tuples bound to *t1* and *t2* may be in different currencies and have different scale factors, they will be compared correctly. In particular, the context mediator will choose values for *Currency* and *ScaleFactor*, convert *t1.Expenses* and *t2.Expenses* to this context, and compare the simple values of the results.

For another example, the following query retrieves the names of those companies trading on the N.Y. stock exchange whose trade price is less than 10 times its dividend:

```
select t.CompanyName
from TRADES t, FINANCES f
where t. Exchange ='nyse' and
        t.TradePrice < 10 * f.Dividend and
        f.CompanyName = t.CompanyName
```

Here, the interesting comparison is between *t.TradePrice* and *f.Dividend*. The intersection of the metaattributes of these two values is {*Currency*}. Because *cvtCurrency* is total, lossless, and order preserving, the comparison is absolute. Thus, the context mediator is free to choose a currency in which to convert the two values before comparing them.

The use of semantic comparison extends directly to all aggregation functions in C-SQL. For example, the aggregate operator *min* in C-SQL differs from the traditional one only in that it uses semantic less than. Thus, in the query

```
select CompanyName
from FINANCES
where Expenses = (select min(Expenses)
                  from FINANCES)
```

the names of the company having the smallest expenses are returned.

Constants in standard SQL queries have no specified context, and thus, the context of a comparison involving a constant is empty. Similarly, if an attribute has no metaattributes, the context of any comparison involving it is also empty. Consequently, such semantic comparisons involve no conversion and are the same as standard (syntactic) comparison in SQL. For example, consider the following query:

```
select CompanyName
from TRADES
where TradePrice > 50
```

Because the context of the comparison is empty, the query retrieves the names of those companies trading higher than 50, regardless of the currency involved. This is rarely the desired effect. If the user wishes to retrieve companies whose trade price has a value greater than 50 U.S. dollars, then a context must be associated with the constant. A natural way to specify this association is to use explicit semantic values; consequently, C-SQL extends SQL so that semantic values can be used as constants. In particular, the appropriate C-SQL query is the following:

```
select CompanyName
from TRADES
where TradePrice > 50(Currency ='USdollars')
```

Here the intersection of the context attributes in the comparison is {*Currency*}, and so semantic comparison gives the desired answer.

C-SQL also extends SQL in that metaattributes can be accessed directly using an extended dot notation. For example, the following query retrieves the company name and trade price (including context) of all stock transactions having a volume of over 56,000 shares whose trades are conducted in Yen:

```
select t.CompanyName, t.TradePrice
from TRADES t
where t.InstrumentType ='equity' and
        t.Volume > 56(ScaleFactor = 1000) and
        t.TradePrice.Currency ='Yen'
```

Note that the comparison on *t.TradePrice.Currency* does not invoke any conversion; only tuples whose value for this attribute is 'Yen' can appear in the answer.

Our stated convention is that the target context of a comparison is determined by taking the intersection of the metaattributes of the items being compared. The rationale behind this convention is that the intersection most closely resemble the generalization of the two items. For example, consider again the comparison of *t.TradePrice* with *f.Dividend*, which appeared in the second query of this section. Intuitively, the generalization of a trade price with a dividend is a money value, which has *Currency* as its single metaattribute.[2] If this convention is not appropriate, then C-SQL provides syntax that allows the user to indicate the target context metaattributes explicitly. In particular, if $E$ is an attribute reference in any part of the query, then $E[A_1, \dots, A_n]$ specifies that all metaattributes apart from the $A_i$ should be projected out of the semantic value. If $n = 0$, then all metaattributes are omitted. For example, suppose that *s1* and *s2* are bound to tuples from *TRADES*. Then the expression *s1.TradePrice*[*Currency*] > *s2.TradePrice*[*Currency*] ignores the price status of the trade prices in the comparison. Such a capability is important for applications that want to eliminate nonresolvable or nonconvertible properties before doing semantic comparison.

The ability to project away metaattributes also allows a user to omit metaattributes specifically from the context of attributes in the **select** clause. For example, consider the following query:

```
select Revenues[ ], Dividend[Periodicity[ ]]
from FINANCES
where CompanyName ='IBM'
```

[2] One could also replace the intersection rule with the use of an explicit generalization hierarchy with properties. For example, the generalization of *TradePrice*(*Currency*, *PriceStatus*) and *Dividend*(*Periodicity*(*FirstIssueDate*), *Currency*) could be defined as *MoneyValue*(*Currency*, *Status*). Then to compare a *TradePrice* and a *Dividend* there must be a mapping between the properties of *Dividend* and *TradePrice* to the property *Status*. Such generalization hierarchies and mappings might be included in the shared ontology.

This query retrieves IBM's revenues without metaattributes and IBM's dividend with only the metaattribute *Periodicity*.

## 5.2 Modifications to Relations

We now consider the treatment of modification commands in the presence of metaattributes. We begin with deletion. Consider the following command:

```
delete from TRADES
where TradePrice < 2(PriceStatus ='latestTradePrice', Currency
        ='USdollars')
```

The semantics of deletion in C-SQL is similar to those of standard SQL, except that semantic comparison is used. Thus, this command deletes those tuples from *TRADES* whose trade price is convertible to a latest trade price of less than 2 U.S. dollars.

As in SQL, insertions in C-SQL can be specified in two ways: The values of the new tuples can be given explicitly (with an associated context if appropriate), or the new tuples can be defined by a selection statement. The following insertion command is an example of the former case:

```
insert into TRADES
values ('IBM', 'equity', 'nyse', 1200(PriceStatus ='latestTradePrice',
          Currency ='Yen'),
          2(ScaleFactor = 100, VolumeStatus ='latestVolume'))
```

An example of the latter case is the following command, which inserts a warrant-instrument tuple into *TRADES* for each traded equity and sets the *TradePrice.PriceStatus* value of these tuples to *'latestTradePrice'*:

```
insert into TRADES
select t.CompanyName, 'warrant', t.Exchange,
        t.TradePrice(PriceStatus ='latestTradePrice', Currency =
        t.TradePrice.Currency)/10,
        t.Volume
from TRADES t
where t.InstrumentType ='equity'
```

The semantics of insertion in C-SQL differs from those of standard SQL in two ways: The first difference is that semantic values and semantic comparisons are used, just as in queries and deletion. The other difference appears when the updated relation contains derived metaattributes. For example, consider the first insertion command above. The semantic-value specification of Figure 4 requires that the *TradePrice* value of the new tuple be stored in U.S. dollars. Consequently, the semantic value *1200(PriceStatus ='latestTradePrice', Currency ='Yen')* must be converted to the proper currency before it is stored. Similarly, the value of *Volume.ScaleFactor* must be converted to 1 before the tuple can be stored.

Intuitively, an insertion to relation *R* works as follows: First, the candidate tuples to be inserted are generated, either by explicit values or by a query;

then these tuples are converted to the environment of $R$ and stored. This process has the following formalization: Let $t$ be a tuple to be inserted into relation $R$. For each attribute $A_i$ of $t$, let $C_{A_i}$ be the context of $t.A_i$. Let $C'_{A_i}$ be the context that results when the metaattribute values in $C_{A_i}$ are changed as required by the environment of $R$, and let $v_{A_i} = cvtVal(t.A_i, C'_{A_i})$. Then the tuple actually inserted into $R$ is $\langle v_{A_1}, \ldots, v_{A_n} \rangle$. If some $v_{A_i}$ does not exist (because a conversion was not possible), then the insertion command is rejected.[3]

In standard SQL, an update to relation $R$ can be viewed as being a two-part process. First, the tuples to be updated are chosen, and then certain attribute values in each tuple are updated. The second part of this process must be altered in C-SQL for exactly the same reasons as insertion. In particular, after the updates specified in the **set** clause are performed, the appropriate attribute values must be converted in order to satisfy the derived metaattribute specifications for $R$. For example, consider the following command:

```
update TRADES
set TradePrice = 9000(PriceStatus ='latestTradePrice', Currency ='Yen')
where CompanyName ='IBM' and Exchange ='nyse'
```

Here, the *TradePrice* value for each tuple denoting a trade on the N.Y. Stock Exchange exchange of IBM is changed to 9000 Yen. However, the environment specifications of Figure 4 require that this value be converted to U.S. dollars in the replaced tuple. Thus, the *TradePrice* value stored is actually the U.S.-dollar equivalent of 9000 Yen.

For another example, consider the following update command:

```
update TRADES
set TradePrice.Currency ='Yen'
where CompanyName ='IBM' and Exchange ='nyse'
```

Let $t$ be a tuple chosen for replacement, and suppose that *t.TradePrice = 9000(PriceStatus ='latestTradePrice', Currency ='USdollars')*. Then the update to the *Currency* metaattribute causes the trade price of $t$ to be interpreted as 9000 Yen. This value is then converted to U.S. dollars (because of the environment requirements) in the updated tuple.

The semantics of update are defined formally as follows: Let $t$ be a tuple of $R$ selected for update. First, the attributes of $t$ (i.e., both base attributes and metaattributes) are changed according to the specifications in the **set** clause; then the tuple is converted back to the environment of $R$ as with insertions. In particular, for each attribute $A_i$ of the updated tuple $t$, let $C_{A_i}$ be the context of $t.A_i$. Let $C'_{A_i}$ be the context that results when the metaattribute

[3] It is important to note that definitions of operational semantics for C-SQL do not imply a plan for how or where the conversion will be done. Similar to query evaluation, conversion may be done on the relation, the predicate, or both. Decisions about conversion strategies will depend on the available routines, their cost, and acceptable orderings. Conversion planning strategies will be examined in future research.

values in $C_{A_i}$ are changed as required by the environment of $R$, and let $v_{A_i} = cvtVal(t.A_i, C'_{A_i})$. Then the final value of the updated tuple is $\langle v_{A_1}, \ldots, v_{A_n} \rangle$. If some $v_{A_i}$ does not exist (because a conversion was not possible), then the update command is rejected.

As a rule, updates to base attributes change attribute values, whereas updates to metaattributes change attribute meanings. Consider the following update command:

```
update TRADES
set Exchange ='madrid'
where CompanyName ='IBM' and Exchange ='nyse'
```

Because the exchange has changed from *'nyse'* to *'madrid'*, the environment requires that the trade price be stored in Pesetas instead of U.S. dollars. Consequently, the value of *TradePrice* will be converted from U.S. dollars to Pesetas when the updated tuple is stored; that is, the new value of *TradePrice* is semantically equal to the previous one. Contrast that command with the following one:

```
update TRADES
set Exchange ='madrid', TradePrice.Currency ='Pesetas'
where CompanyName ='IBM' and Exchange ='nyse'
```

Here, the value of *TradePrice.Currency* is being changed explicitly, and so no conversion will be done when the updated tuple is stored. That is, if the *TradePrice* value of a tuple was 35 U.S. dollars, then it will be changed to 35 Pesetas.

## 5.3 Derived Relations

One of the important features of the relational model is that the result of a query is a relation, and can be stored as a snapshot relation or defined as a view. We call a snapshot or a view a *derived relation*. The one complexity caused by the presence of metaattributes is how the schema of the derived relation is defined.

In standard SQL, the schema of a derived relation is determined by the attributes in the **select** list. If a different attribute name is desired in the derived relation from the one appearing in the **select** list, then the **as** keyword can be used to do the renaming. A similar capability exists in C-SQL. In C-SQL, however, this usage is extended so that attributes can be renamed as metaattributes and vice versa, thereby allowing attributes in the **select** list to be restructured. For example, consider the following:

```
create relation COMPANY-EXCHANGE as
select t.CompanyName as CompanyName,
       f.Dividend.Currency as CompanyName.Currency,
       t.Volume[ ] as Volume,
       t.Volume.ScaleFactor as ScaleFactor
from TRADES t, FINANCES f
where f.CompanyName = t.CompanyName
```

The derived relation *COMPANY-EXCHANGE* has the three base attributes *CompanyName*, *Volume*, and *ScaleFactor*; there is also one metaattribute, namely, *CompanyName.Currency*. Note that, unlike the underlying database, in the derived relation the attribute *CompanyName* has a metaattribute and *ScaleFactor* is a base attribute.

Derived relations can be included in the semantic-value specification of an information system. For a simple example, suppose that the source database defines the view

```
create view US-TRADES as
select *
from TRADES
```

and suppose that the semantic-value specification of the source contains the following scene:

```
create scene for US-TRADES by predicate
  Currency ='USdollars' and ScaleFactor = 1000
```

The derived relation *US-TRADES* contains the same tuples as *TRADES*; the only difference is that the tuples have a different context. The benefit of such a view definition is that users accessing it can assume that all tuples have the same currency (i.e., U.S. dollars) and the same scale factor (i.e., 1000); this regularity makes it easier to formulate queries. Intuitively, *US-TRADES* provides a "semantic view" of the relation *TRADES*, in the sense that the user sees the same data, but the data have a different representation.

In general, the calculation of a derived relation $V$ proceeds in two steps. First, any scenes for $V$ are ignored, and the underlying query is executed, resulting in a temporary relation $T$. Second, an empty relation for $V$ is created, and the tuples of $T$ are inserted into $V$; the scene $V$ causes the tuples to be converted appropriately, as shown in Section 5.2.

For example, consider the following view definition:

```
create view SMALL-COMPANY as
select *
from FINANCES
where Revenues < 1000(Currency ='USdollars', ScaleFactor = 1000)
```

The view *SMALL-COMPANY* contains those *FINANCES* tuples having revenues less than 1,000,000 U.S. dollars. However, the tuples in the view still have their original currencies and scale factors, and queries over the view would have to take that into consideration. Suppose, however, that we also define the following environment for the view:

```
create scene for SMALL-COMPANY by predicate
  Currency ='Yen' and ScaleFactor = 1
```

The effect of this scene specification is to ensure that all tuples in the view are seen in the specified contexts. That is, the view still contains those *FINANCES* tuples having revenues less than 1,000,000 U.S. dollars; the values of the tuples are just seen as if they were all stored in Yen and unscaled.

As in the standard relational model, derived relations are accessed just like base relations. In particular, updates to views have the same restrictions as in the relational model. For example, consider the following insertion:

> **insert into** *SMALL-COMPANY*
> **values** ('Batik Bali', 'Denpasar'(locationGranularity ='city'),
> *300000(Currency ='Indonesian Rupiah'),*
> *260000(Currency ='Indonesian Rupiah'), 0)*

This new tuple asserts that the Batik Bali company has revenues of 300,000 Rupiah and expenses of 260,000 Rupiah. Its insertion proceeds as follows: The currency values of the tuple are first converted to Yen in order to insert it into *SMALL-COMPANY*. Then that converted tuple is inserted into *FINANCES*; this tuple is then converted according to the data environment of that relation, and finally stored.

As we have seen, the way to change the context of tuples in the database is to create a derived relation to contain the tuples and to associate a scene with it. Scenes can also be used to given values to new metaattributes in derived relations. For example, consider again the derived relation *COMPANY-EXCHANGE* introduced at the beginning of this subsection. The new metaattribute *CompanyName.Currency* was given the value of *f.Dividend.Currency*. Suppose now that the notion of currency did not appear in either *TRADES* or *FINANCES*. Then it would still be possible to give *CompanyName.Currency* a value by using a scene. In particular, the line for *CompanyName.Currency* in the **select** clause of the snapshot definition would be changed to contain the line "**null as** *CompanyName.Currency*," and the following scene could be added:

> **create scene for** *COMPANY-EXCHANGE* **by predicate**
> *CompanyName.Currency ='USdollars'*

### 5.4 The Receiver's Data Environment

We have been assuming that the data environment of the receiver is the same as the data environment of the source. In this subsection we consider the case where the semantic-value specification of the receiver differs from that of the source. Additionally, we investigate the related problem of how to associate environments with queries and updates.

A receiver does not see the source data directly; instead, all values are filtered through the receiver's data environment. Formally, the receiver sees a *view* of each source relation. If the semantic-value specification of the receiver is the same as that of the source, as we had been assuming, then the values seen by the receiver are the same as they appear in the source. If the semantic-value specification of the receiver is different from the source, then the view contains the same values as the source, but the context of the values will be different. In particular, the receiver implicitly defines a view for each source relation, and the scenes in the receiver's data environment are interpreted as referring to these views instead of to the source relations. For example, suppose that the receiver contains the following semantic-value specification in its data environment:

```
create scene for TRADES by predicate
  Currency ='USdollars' and ScaleFactor = 1000
```

The effect of this scene is that the receiver implicitly defines a local view called *TRADES*, having the same tuples as the source relation *TRADES*; the scene then applies to the view instead of to the source relation. Consequently, the relations *TRADES* and *US-TRADES* (as defined in the previous subsection) look exactly the same to the receiver. This approach to modeling the receiver's data environment extends [Siegel and Madnick 1991], where these data environments are called *Application Semantic Views*.

Queries and updates are posed from within the receiver's data environment and, thus, are affected by it. It is also possible for a request to be made from a different perspective than what appears in the receiver's data environment. In order to handle such a case, we introduce into C-SQL a new language construct, called the **inContext** clause, in which a scene is specified for an entire request.

For an example, suppose that a user wishes to retrieve all *TRADES* tuples having a latest trade price of more than 15 U.S. dollars, and wishes to view their values in U.S. dollars as well. Then one solution would be to define a view of *TRADES* in which all currencies are converted to U.S. dollars, and then to perform a query on the view. The following query uses the **inContext** clause to do the same thing in a single step:

```
select *
from TRADES
inContext Currency ='USdollars' and PriceStatus ='latestTradePrice'
where TradePrice > 15
```

Formally, the **inContext** clause affects a query or update request in the following way: First, a temporary (cartesian product) view is generated using the contents of the **from** clause. Second, the **inContext** clause is used to define a scene for the view. Third, the **where** clause is evaluated over the view. Thus, in the above query, all trade prices are converted to latest trade prices in U.S. dollars in the temporary view; the predicate *TradePrice > 15* therefore selects those tuples having a latest trade price over 15 U.S. dollars.

Queries can be nested, and each nested subquery can have its own scene. For example, the following query returns the name and trade price in U.S. dollars of all companies whose revenues are more than 500,000 Yen:

```
select CompanyName, TradePrice
from TRADES
inContext Currency ='USdollars'
where CompanyName in (select CompanyName
                      from FINANCES
                      inContext Currency ='Yen' and ScaleFactor =
                                1000
                      where Revenues > 500)
```

We are now at a point where we can discuss relative semantic comparison. Recall from Section 2.4 that relative semantic comparison is meaningful in C-SQL if the semantic values being compared have exactly the same context. Although this condition is restrictive in general, a proper use of the **inContext** clause makes relative semantic comparison straightforward. For example, consider the following query:

**select** *t1.CompanyName*
**from** *FINANCES t1 t2*
**inContext** *LocationGranularity* ='*country*'
**where** *t1.Location* = *t2.Location* **and** *t2.CompanyName* ='*IBM*'

Here, the nontotal comparison involves *Location*. The scene of the query asserts that this comparison should be performed in the context *LocationGranularity* ='*country*'; that is, the query retrieves the names of those companies located in the same country as IBM. Note that the query is executed in an environment in which *LocationGranularity* is assigned a specific value. Consequently, all semantic values for *Location* will have the same context, and so relative semantic comparison will be possible. In effect, the scene acts as the specification of the target context of the comparison.

We began this paper by developing a model-independent theory for the exchange of information using semantic values. In this and the previous section, we have shown how this theory can be applied to the relational model. In the next section, we describe a prototype system that we are in the process of developing that implements a subset of the capabilities described in this section.

## 6. IMPLEMENTATION OF A CONTEXT MEDIATOR

In this section we examine one particular implementation of a context mediator. Our discussion is necessarily brief; a full discussion can be found in Madero [1992]. This context mediator assists in the exchange of semantic values between a relational application as data receiver and a relational database as data source, using a standard SQL interface and rule-based data environments. In such a system, the representation and manipulation of semantic values remain transparent to the user.

There are several assumptions made in this implementation. First, we assume that the relations provided by the database and viewed by the application have exactly the same base attributes; the only difference between the relations is that different metaattributes and metaattribute values may be specified by the database and application. Second, we assume that all metaattributes are derived and that data environments are rule based, with the antecedent of each rule composed of restrictions on base attributes and the consequent assigning values to metaattributes; detailed discussions of the structure of rules appear in Madero [1992] and Siegel and Madnick [1991]. Third, we assume that the database and application agree on the meaning of attribute names, properties, and values, and so the terminology mappings of Section 3 are not needed. Finally, we assume that conversion functions follow the naming conversions of this paper and are stored centrally with the context mediator.[4]

As a running example, we assume that the semantic-value schemata for both the application and the database are the semantic values in the *TRADES* relation of Figure 2; the database uses the data environment of Figure 4, and the application uses the data environment of Figure 5.

The context mediator's first step, before it examines any query, is to preprocess the data environments of the database and the application. The purpose of this preprocessing stage is for the context mediator to become aware of potential conflicts in queries and to record the conversion functions (if any) needed to remove the conflict. During this stage, the context mediator compares each rule from the application with each rule in the database, looking for pairs of *conflicting* rules. We say that two rules conflict if their antecedent have a nonempty intersection (i.e., it is possible for a single tuple to satisfy the conditions of both antecedents) and if their consequents assign different values to the same metaattribute. The test for nonempty intersection of antecedents is called *subsumption* and is discussed in Madero [1992] and Siegel and Madnick [1991]. The subsumption process also determines the condition under which the conflict will occur. Information about conflicts are stored in the *conflict table*. Each row in the conflict table describes the condition causing a conflict, the attribute and metaattribute involved, the conflicting values assigned to the metaattribute, and the conversion function (if any) required to resolve the conflict.

Consider the data environments in our running example. The first rules from both data environments have a nonempty intersection, namely, all equities traded on the Madrid Stock Exchange; moreover, the consequents of these rules conflict, because the database rule assigns the value of *TradePrice.Currency* to be in Pesetas, whereas the application requires the currency to be in U.S. dollars. Thus, these two rules conflict. There happen to be four conflicting rule pairs in this example, giving rise to the conflict table of Table I. Rule 3 of the database environment conflicts with rule 2 of the application environment over the metaattribute *TradePrice.PriceStatus*, giving rise to the second row of the conflict table. The third row of the conflict table results from rule 4 of the database environment and rule 3 of the application environment, and the fourth row of the conflict table results from rule 5 of the database environment and rule 3 of the application environment.

Intuitively, each row of the conflict table describes how a tuple from the source database should be converted to the context of the receiving applica-

[4] Relaxing these assumptions is possible [Siegel and Madnick 1991], but significantly increases the complexity of implementation. For example, if the database and application have different schemata, then an ontological component is needed to provide the context mediator with terminology mappings. Moreover, it is often the case that such a restriction is reasonable in practice. If the receiver mentions a metaattribute that does not appear in the source, then semantic comparisons involving that metaattribute become unresolvable. For example, suppose that the receiver's environment defines *TradePrice* with metaattribute *Currency*, but that no such metaattribute appears in the source's environment. Then there may be no way to determine the currency of a source's value.

```
create scene for TRADES by rules
    if Exchange = 'madrid'
        then TradePrice.PriceStatus = 'latestClosingPrice' and
                TradePrice.Currency = 'US dollars';
    if Exchange = 'nyse'
        then TradePrice.PriceStatus = 'latestTradePrice' and
                TradePrice.Currency = 'USdollars';
    if TRUE
        then Volume.VolumeStatus = 'closingVolume' and
                Volume.Scalefactor = 1;
```

Fig. 5. Application's data environment.

tion. In particular, let $t$ be a source tuple. If $t$ satisfies none of the constraints of the rows in the conflict table, then $t$ can be passed to the receiver without any conversions. If $t$ satisfies one or more of the constraints, however, then appropriate conversions must occur, according to the specified conversion functions. For example, suppose that $t$ has the values *t.InstrumentType* =*'equity'* and *t.Exchange* =*'madrid'*. Then $t$ satisfies the constraints of the first and third rows of the conflict table; so *t.TradePrice* must be converted from Pesetas to U.S. dollars, and *t.Volume* must be converted from the latest volume to the closing volume. However, note that *t.Volume* cannot be converted, because the third row of the conflict table has no associated conversion function. Thus, for any typle $t$ satisfying the constraint of this row, the value of the attribute *t.Volume* cannot be meaningfully sent to the receiver or used to answer a query. The context mediator will either abort or return a partial answer if permitted by the application.

After the preprocessing stage is finished, the context mediator is ready to process queries from the application. Recall from Section 3.2 that a context mediator can perform semantic comparisons in a variety of contexts. Currently, our context mediator always applies semantic comparisons in the context of the receiver (i.e., the target context). When a query $Q$ is submitted to the context mediator, the first thing it does is to place the predicate in the **where** clause into conjunctive normal form (CNF). It then performs the following four steps:

*Step* 1. Determine those attributes mentioned in $Q$ whose values will need to be converted; we call such attributes *unsafe*. The unsafe atributes are found by examining each row of the conflict table. If a row's constraint overlaps with the predicate of $Q$'s **where** clause and the attribute associated with the row appears in $Q$, then that attribute is unsafe. A conjunct containing an unsafe attribute is called an *unsafe conjunct*.

*Step* 2. Create an intermediate query $Q_R$ for each relation $R$ in $Q$'s **from** list.[5] The purpose of these intermediate queries is to retrieve as small as possible portion of the database to convert to the target context. The **where**

[5] More accurately, a query is created for each tuple variable in the query.

Table I. Conflict Table

| Constraint | Attribute | Metaattribute | Database value | Application value | Conversion function |
|---|---|---|---|---|---|
| *InstrumentType* =‘*equity*’<br>**and** *Exchange* =‘*madrid*’ | *TradePrice* | *Currency* | ‘*Pesetas*’ | ‘*USdollars*’ | *cvtCurrency* |
| *InstrumentType* =‘*future*’<br>**and** *Exchange* =‘*nyse*’ | *TradePrice* | *PriceStatus* | ‘*latestClosingPrice*’ | ‘*latestTradePrice*’ | None |
| *InstrumentType* =‘*equity*’ | *Volume* | *VolumeStatus* | ‘*latestVolume*’ | ‘*closingVolume*’ | None |
| *InstrumentType* =‘*future*’ | *Volume* | *ScaleFactor* | *1000* | *1* | *cvtScaleFactor* |

clause of $Q_R$ contains those safe conjuncts from $Q$ mentioning only attributes of $R$. The **select** clause of $Q_R$ contains those attributes of $R$ appearing in $Q$'s **select** list, those attributes of $R$ mentioned in unsafe conjuncts or join conditions, and those attributes of $R$ needed to determine the context of its unsafe attributes using the conflict table.

*Step* 3. Send each query $Q_R$ to the source database. For each tuple returned, the conflict table is used to determine the appropriate conversions, and the converted tuple is saved in a temporary relation $R'$. If one or more tuples cannot be converted, then the context mediator has two courses of action, depending on the wishes of the application: It can abort the query, or it may omit the nonconvertible tuples from the result.

*Step* 4. Evaluate the query $Q'$ on the temporary relations, where $Q'$ is constructed as follows: The **select** clause of $Q'$ contains the same attributes as that of $Q$; the **from** clause of $Q'$ contains the same items as that of $Q$, except that each reference to $R$ is replaced by $R'$; and the **where** clause of $Q'$ contains those conjuncts from $Q$ that were omitted from all of the $Q_R$'s (i.e., the unsafe conjuncts and join conditions). The tuples produced from this query are sent to the receiver.

For an example of this process, suppose that the application poses the following query:

**select** *CompanyName, TradePrice*
**from** *TRADES*
**where** *InstrumentType* =*'equıty'* **and** *TradePrice* > *100*

Because the application uses the data environment of Figure 5, this query requests the name and trade price of those equities having a latest closing price of more than 100 U.S. dollars. Noting that the query is already in CNF, the context mediator proceeds as follows.

In Step 1, the predicate "*InstrumentType* =*'equity'* **and** *TradePrice* > *100*" is intersected with the condition in each row of the conflict table, and a nonnull intersection is found for the first and third rows. These two rows have corresponding attributes *TradePrice* and *Volume*; of these, only *TradePrice* appears in the query and, thus, is the only unsafe attribute. The following query $Q_R$ is constructed for Step 2.

**select** *CompanyName, TradePrice, Exchange*
**from** *TRADES*
**where** *InstrumentType* =*'equity'*

Note that the attribute *Exchange* is selected, because it is needed to determine the context of the relation's *TradePrice* values. In Step 3, query $Q_R$ is sent to the database, and the received tuples are converted. In particular, the *TradePrice* value of each tuple traded on the Madrid Stock Exchange is converted from Pesetas to U.S. dollars. Any tuples in $Q_R$ having an *Exchange* value of *'nyse'* cannot be converted. The converted tuples are stored in the temporary relation *TRADES′*. Finally, the following query is evaluated in Step 4:

**select** *CompanyName*, *TradePrice*
**from** *TRADES′*
**where** *TradePrice* > *100*

and the resulting tuples are sent to the receiver.

In our implementation, Step 2 is refined further in order to improve the generation of conversion plans. In particular, instead of constructing $Q_R$ directly, the context mediator uses the conflict table to determine a set of conditions that partition $Q_R$; each partition corresponds to a conversion plan. A query is then generated for each of these partitions. The tuples returning from each of these queries are converted in Step 3 according to the conversion plan for that query, and the union of these converted relations is stored in the temporary relation $R'$. For example, the above example query $Q_R$ has two partitions, corresponding to the following queries:

**select** *CompanyName*, *TradePrice*
**from** *TRADES*
**where** *InstrumentType* =*'equity'* **and** *Exchange* =*'madrid'*

**select** *CompanyName*, *TradePrice*
**from** *TRADES*
**where** *InstrumentType* =*'equity'* **and** *Exchange* ≠*'madrid'*

Each tuple $t$ from the first query will be converted so that *t.TradePrice.Currency* =*'USdollars'*; tuples from the second query need no conversion.

One advantage of performing this partitioning is that the context mediator can generate a conversion plan once and apply it in bulk. Another advantage is that impossible conversions can be determined quickly, without having to go to the database. For example, the trade price for a future traded on the N.Y. Stock Exchange could not be converted from the database context to the application context because there is no conversion function for the *PriceStatus* metaattribute.

This use of constraints to analyze a query resembles work in semantic query optimization [Chakravarthy et al. 1986; King 1981; Siegel et al. 1992; Yu and Sun 1989]. In general, the context mediator may (1) identify semantic comparisons that may never be true, thus eliminating further analysis; (2) identify semantic comparisons that are always true, thus reducing the need for run-time conversions; (3) resolve queries through contradiction; and (4) make it possible to simplify queries so that the data environment provides the answer to it. As with semantic query optimization [Chakravarthy et al. 1986; King 1981; Reddy et al. 1992a; Siegel et al. 1992; Yu and Sun 1989], this preprocessing can considerable reduce query-processing costs.

The steps described for the context mediator are not intended to define an optimal plan for query processing using semantic values. Optimization issues in context mediation is being examined as part of our current research effort. The existing context mediator runs under UNIX,[TM] is written in C, and uses

[TM] UNIX is a trademark.

a relational database to store intermediate results. Current plans include adding a C-SQL parser, building a context mediator for a C-SQL application possibly as part of a hierarchy of cooperating mediators, and using more general data environments. We also are examining different approaches to subsumption such as the use of a Datalog or logic processing engine [Collet et al. 1991; Shen et al. 1991].

## 7. CONCLUSIONS

This research has provided a model-independent theory for the exchange of data among heterogeneous information systems. Our approach is to use semantic values as the unit of exchange and to have a context mediator to facilitate this exchange. We use data environments as a way for the component information systems to describe the values of the properties of the semantic values they exchange. We then apply these ideas to the relational model, developing an SQL-based data manipulation language, called C-SQL. Finally, we describe a prototype where rule-based data environments are used to hide all context information from the application. Application requests are expressed in SQL, and the context mediator guarantees the correct semantics for the result by utilizing the application and database environments.

Much of the previous work in semantic interoperability was developed for static systems and presented itself as monolithic; some suggested a single integrated interface (i.e., global schema) having a single environment [Jakobson et al. 1988; Landers and Rosenberg, 1992; Siegel et al. 1991; Templeton et al. 1987], others a single data model with multiple environments [Litwin and Abdellatif 1986; 1987], and others a federated approach with each member of the federation working in a different environment [Sheth and Larson 1990]. Our approach is more dynamic, expressive, and extensible. It allows for a component information system to change its environment, either for external reasons or as a reaction to the change in the environment of another system. We believe that our approach is complementry to these other approaches and can be incorporated into any of them. For example, in the integrated approach, multiple-component systems will share a combined view having a single data environment; in the federated approach, the component systems may be accessed by explicitly sharing the necessary semantic values. Current research efforts are investigating the integration of our architecture for context mediation with developing multidatabase architectures [Reddy et al. 1992b; Rosenthal and Siegel 1991] to address issues in semantic interoperability including multidatabase query optimization [Reddy et al. 1992a].

We improve on current technology by allowing gradual, modular development of data descriptions and conversion functions. Instead of requiring each pair of communicating systems to determine an interface, the context mediator compares environments and synthesizes the translations. Context mediators are designed to be fixed components, changing only when the interface changes, whereas the environments, ontologies, and conversion routines are expected to be evolutionary. The ability to interchange can grow gradually, semantic value by semantic value, instead on in larger steps requiring understanding of the systems performing the interchange.

Our work suggests several lines for future research. First, context information needs to be attached to objects larger than single attributes; this may be easier in object-oriented models. Second, we need to examine a large range of conversion functions from a wide variety of application domains in order to verify the applicability of our theory. The properties of conversion functions (e.g., losslessness) and the meaning of semantic comparison need to be defined in terms of imprecise conversion. For example, in assuming the losslessness of *cvtCurrency* and *cvtLengthUnit*, we have ignored possible roundoff error from the conversion of values. Similarly, we need to examine representation, comparison, and conversion techniques for complex context information such as derived data formulas. Temporal issues must also be considered, as historical databases are likely to have context that changes over time. Third, we need to consider the possibility of using multiple, cooperating mediators. Additionally, considerable research is needed to develop strategies for building mediators [Neches et al. 1991]. It is necessry to define algorithms that the mediator can use to plan conversions; this planning can be nontrivial when conversions' behavior is more complex than simply changing a metaattribute from one value to another. Fourth, optimization techniques are needed for planning the evaluation of queries; for example, we believe that many of the results in Siegel and Madnick [1991] can be restated in our extended relational model as certain forms of semantic query optimization. Finally, it is necessary to understand better the tools, architectures, and organizational requirements that would allow data administrators and application developers to cope with context interchange in a large-scale environment (e.g., ontologies, conversion libraries, common knowledge representation languages). We continue to develop the model-independent theory, as this will simplify the use of this approach to a range of data and system models.

## ACKNOWLEDGMENTS

The authors would like to thank Stuart Madnick for his endless insight into problems related to integration, semantic interoperability, and context interchange.

## REFERENCES

CHAKRAVARTHY, U., FISHMAN, D., AND MINKER, J. 1986. Semantic query optimization in expert systems and database systems. In *Expert Database Systems: Proceedings of the 1st International Workshop*, L. Kershberg, Ed. Benjamin Cummings, Menlo Park, Calif., 659–674.

COLLET, C., HUHNS, M., AND SHEN, W. 1991. Resource integration using an existing large knowledge base. Tech. Rep. ACT-OODS-127-91, MCC, Austin, Tex.

JAKOBSON, G., PIATETSKY-SHAPIRO, G., LAFOND, C., RAJINIKANTH, M., AND HERNANDEZ, J. 1988. CALIDA: A system for integrated retrieval from multiple heterogeneous databases. In *Proceedings of the 3rd International Conference on Data and Knowledge Engineering* (Jerusalem, Nov.). 3–18.

King, J 1981. QUIST: A system for semantic query optimization in relational databases In *Proceedings 7th International Conference on Very Large Data Bases* (Cannes, France). VLDB Endowment, 510–517.

Landers, T., and Rosenberg, R. 1982. An overview of multibase. In *Distributed Data Bases*, H. J. Schneider, Ed. North-Holland, Amsterdam, 153–183.

Litwin, W., and Abdellatif, A. 1987. An overview of the multidatabase manipulation language mdsl. *Proc. IEEE 75*, 5 (May), 621–631.

Litwin, W , and Abdellatif, A. 1986. Multidatabase interoperability *Computer 19*, 12 (Dec.), 10–18.

Madero, F. 1992. Rule-based mediator implementation for solving semantic conflicts in SQL. Master's thesis, MIT, Sloan School of Management, Cambridge, Mass., Sept.

McCarthy, J. 1984. Scientific information = data + meta-data. Database Management Proceedings of the Workshop November 1–2, U.S. Navy Postgraduate School, Monterey, California. Tech. Rep., Dept. of Statistics. Stanford Univ., Calif.

Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T., and Swartout, W. 1991. Enabling technology for knowledge sharing. *AI Mag. 12*, 3 (Mar ), 37–56.

Reddy, M. P , Siegel, M., and Gupta, A. 1992a. Semantic query processing in HDDBMS. *VLDB J.* To be published.

Reddy, M. P., Siegel, M., and Gupta, A. 1992b. Towards an active schema integration architecture for heterogeneous database systems. In *International Workshop on Research Issues on Data Engineering: Interoperability in Multidatabase Systems (RIDE-IMS93)* (Vienna, Austria). IEEE, New York, 178–183.

Rosenthal, A., and Siegel, M. 1991. An architecture for practical metadata integration. In *Workshop on Information Technologies Systems* (Cambridge, Mass , Dec ). 98–106.

Sciore, E., Siegel, M., and Rosenthal, A. 1992. Context interchange using meta-attributes In *1st International Conference on Information and Knowledge Management* (Baltimore, Md ) ACM, New York, 377–386.

Shen, M., Huhns, W., and Collet, C. 1991. Resource integration without application modification. Tech. Rep. ACT-OODS-214-91, MCC, Austin, Tex.

Sheth, A., and Larson, J. 1990. Federated databases: Architectures and integration. *ACM Comput. Surv. 22*, 3 (Sept.), 183–236

Siegel, M , and Madnick, S. 1991. A metadata approach to resolving semantic conflicts. In *Proceedings of the 17th International Conference on Very Large Data Bases* (Barcelona, Spain, Sept.). VLDB Endowment, 133–145

Siegel, M., and Madnick, S. 1989. Schema integration using metadata. In *Position Papers: NSF Workshop on Heterogeneous Databases* (Evanston, Ill., Dec. 11–13).

Siegel, M., Madnick, S., and Gupta, A. 1991 Composite information systems Resolving semantic heterogeneities. In *Workshop on Information Technology Systems* (Cambridge, Mass , Dec.). 125–140.

Siegel, M., Salveter, S., and Sciore, E. 1992. Automatic rule derivation for semantic query optimization. *ACM Trans. Database Syst. 17*, 4 (Dec.), 563–600.

Siegel, M., Sciore, E , and Madnick, S 1993. Context interchange in a client-server architecture. *J. Softw. Syst.* To be published.

Templeton, M., Brill, D., Hwang, A., Kameny, I., and Lund, E. 1987. Mermaid—A front-end to distributed heterogeneous databases. *Proc. IEEE 75*, 5 (May), 695–708.

Wang, R., ed. 1992. *Information Technology in Action: Trends and Perspectives.* Prentice-Hall, Englewood Cliffs, N.J.

Wang, R., and Madnick, S. 1990. Data-source tagging. In *Proceeding from the 16th Very Large Data Base Conference* (Brisbane, Australia). VLDB Endowment, 519–538.

Wiederhold, G. 1992. Mediators in the architecture of future information systems. *Computer 25*, 3 (Mar.), 38–49.

Yu, C., and Sun, W. 1989. Automatic knowledge acquisition and maintenance for semantic query optimization. *IEEE Trans. Knowl. Data Eng. 1*, 3 (Sept.), 362–375.

Received December 1992; revised June 1993; accepted October 1993