A Logical Semantics for Object-Oriented Databases

José Meseguer*and Xiaolei Qian†
Computer Science Laboratory, SRI International
333 Ravenswood Avenue, Menlo Park, CA 94025, USA

Abstract

Although the mathematical foundations of relational databases are very well established, the state of affairs for object-oriented databases is much less satisfactory. We propose a semantic foundation for object-oriented databases based on a simple logic of change called *rewriting logic*, and a language called MaudeLog that is based on that logic. Some key advantages of our approach include its logical nature, its simplicity without any need for higher-order features, the fact that dynamic aspects are directly addressed, the rigorous integration of user-definable algebraic data types within the framework, the existence of initial models, and the integration of query, update, and programming aspects within a single declarative language.

1 Introduction

Although the mathematical foundations of relational databases are very well established, the state of affairs for object-oriented databases is much less satisfactory. This is unfortunate, because object-oriented databases seem to have important advantages over competing approaches; however, the problem is not unique to databases and encompasses also object-oriented programming languages that are equally in serious need of semantic foundations. In fact, the related problem of the "impedance mismatch" between databases and programming languages can be interpreted as another symptom of the same lack of semantic foundations, since the appropriate integration of databases and programming languages is also in need of such foundations.

The need for semantic foundations in object-oriented databases has been felt quite strongly by researchers and has led to a number of attempts and proposals, including for example work by Maier on O-logic [26], work by Kifer and Lausen on F-logic [23], work by Goguen, Meseguer and Wolfram on the FOOPS language [17, 21], work by Beeri and Milo on algebraic foundations [8, 9, 10], and work by Abiteboul on method schemas [4, 3] (see [23, 9] for additional references in this whole area). However, it seems fair to say that no agreement has yet been reached on the matter of foundations, and that many problems remain open as serious challenges, especially in dynamic aspects having to do with evolution in time and state change.

We propose a semantic foundation for object-oriented databases based on a simple logic of change called *rewriting logic*, and a language called MaudeLog that is based on that logic. Some key advantages of our approach include its logical nature, its simplicity without any need for higher-order features, the fact that dynamic aspects are directly addressed, the rigorous integration of user-definable algebraic data types within the framework, the existence of initial models, and the integration of query, update, and programming aspects within a single declarative language. Our work builds on previous work by the first author on the semantic foundations of object-oriented programming using rewriting logic [27, 29], and can be viewed as a natural extension of that work to object-oriented databases.

To give the reader a better feeling for how some of the key semantic issues—such as behavioral aspects, evolution in time, impedance mismatch between databases and programming languages, inheritance, higher-order capabilities, and views—are addressed in our approach, we briefly comment on those aspects below. In the rest of the paper, we informally introduce MaudeLog and the rewriting formalism with examples in Section 2, we explain the basic ideas about rewriting logic in Section 3, and then we give a more formal discussion of object-oriented database concepts in Section 4. We end the paper with some concluding remarks about directions for future work.

Behavioral Aspects Beeri [9] has commented that behavioral aspects have not been considered in depth by the database theory community. Initial efforts in this area include the work on F-Logic [23], and work on method schemas [4, 3], which uses a form of graph rewriting.

In MaudeLog, type structure and behavior are fully integrated in an implementation-independent way. The type structure is made up of user-definable algebraic data types, object classes (with objects having attributes and being queried and updated by messages[1]), and subtype (for data) and class (for objects) hierarchies. Behavior is specified by

*Supported in part by Office of Naval Research under Contracts N00014-90-C-0086 and N00014-92-C-0518, and by the Information Technology Promotion Agency, Japan, as part of the R & D of Basic Technology for Future Industries "New Models for Software Architecture" sponsored by NEDO (New Energy and Industrial Technology Development Organization).

†Supported in part by Defense Advanced Research Projects Agency and Rome Laboratory under Contract F30602-92-C-0140.

SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0089...$1.50

[1] A *message* is sometimes called a *method invocation*. We

functions of algebraic data types and by messages for basic and derived, or computed, attributes in objects. Behavior is captured by deduction in equational (for data and functions) and rewriting (for objects and messages) logics.

Evolution in Time Evolution in time (state change) is formalized in MaudeLog through rewriting logic. Databases are conceptualized as concurrent *systems* of active objects that evolve by manipulating attributes and exchanging messages. Object creation, deletion, and uniqueness of object identity are also supported by the logic [29]. MaudeLog captures the object-oriented paradigm—including evolution in time—in a fully declarative way.

In comparison, the relational model conceptualizes databases as *sets* of objects, which captures *structural* aspects of objects. Existing object-oriented data models conceptualize databases as *algebras* of objects, which capture *behavioral* aspects of objects. No other data modeling formalisms that we are aware of capture evolution in time (see [9]). In particular, existing approaches do not handle updates, they cannot model the fact that object identity does not change even when its value is updated, and object generation remains a challenge [9].

Impedance Mismatch As pointed out by Beeri [9], most object-oriented database systems support behavioral aspects only procedurally. F-Logic [23] advocates a declarative style to program methods, but its computation is limited to derived attributes and does not seem to address database updates. Existing database languages do not fully support the declarative programming of updates. For example, IQL [2] uses a mixture of declarative and procedural styles, and does not support the deletion of objects.

MaudeLog solves the impedance mismatch problem between databases and programming languages. It is not just an object-oriented data modeling formalism, but also a complete object-oriented query, update, and programming language. It supports declarative programming with collection (bulk) types independent of specific type system implementations.

Inheritance MaudeLog's distinction between class inheritance—used only for the purpose of classifying objects in taxonomic class hierarchies—and module inheritance—used to reuse code in modules—avoids the problems associated with burdening the class inheritance mechanism with the task of supporting code reuse. Module inheritance's support for modularity, parameterization, and encapsulation greatly facilitates object-oriented programming in the large and a highly modular style of schema evolution.

First-order vs. Higher-order Many database researchers believe that first-order logic is inadequate as a formal basis for object-oriented databases, and that higher-order logics are necessary [9]. Examples are HiLog [14] and F-Logic [23]. However, the extra complexity brought by being higher-order is well recognized. For example, in order to have a first-order semantics, F-Logic has to impose a static lattice ordering on the universe of discourse.

Although rewriting logic is a logic of change and as such it is remarkably different from first-order logic, it nevertheless shares with first-order logic not involving any higher-order features. However, higher-order capabilities are available thanks to parameterization and module inheritance mechanisms, without any need for the semantic framework itself being higher-order. In particular, meta data is dealt with using module hierarchies, parameterized modules, module expressions, and theory interpretations. Since meta data is dealt with at the module level and is therefore cleanly separated from data, there is no need for introducing higher-order features; this makes the formal approach much simpler than higher-order approaches.

Views One reason why many researchers believe that higher-order logics are necessary for object-oriented databases is the need for supporting views. Following the relational model, object-oriented views have been formulated as queries [1, 11]. Therefore, query languages have been burdened with the requirement of expressing views, which leads to the argument for query languages being higher-order.

In MaudeLog, views are closely related to theory interpretations, of which the relational views are a special case. Therefore, MaudeLog supports object-oriented views without any need for higher-order logics. This is very much in the spirit of Goguen [16] who, in the context of algebraic specifications, first advocated theory interpretations to handle database views. Due to space limitations, discussions on the details of the view mechanism in MaudeLog are left for another paper.

2 OO Data Modeling

This section provides an informal introduction with examples to MaudeLog, illustrating how MaudeLog supports a modular and declarative approach to the specification of object-oriented databases that includes both static and dynamic aspects.

2.1 Schemas and Databases

A schema consists of modules organized into hierarchies. There are two kinds of modules[2], namely functional modules—which give rise to a functional sublanguage almost identical to OBJ3 [20]—and object-oriented modules.

Functional modules support user-definable algebraic data types as part of the schema and therefore the ability of incorporating a very rich, extensible, collection of data types within a database. The importance of algebraic specification for specifying schemas and for supporting a functional style of database computations has been emphasized by previous research, such as for example [16, 9, 10]; it is also closely related to the topic of "collection" or "bulk" types [7, 12].

Object-oriented modules support the declarative definition of class hierarchies of objects that can be updated

prefer the message terminology to emphasize the potentially distributed nature of object-oriented computation.

[2] System modules, of which object-oriented modules are a special case, are not discussed in this paper; see [29].

and queried in a distributed fashion by means of messages. Therefore, all the advantages of object-orientation for structuring, classifying and querying data are available. One of the key objectives of this paper, to be further discussed in Sections 3 and 4, is precisely to propose a simple and rigorous foundation of object-oriented databases that, in addition, integrates nicely the functional perspective.

2.1.1 Functional Modules

A *functional module* begins with the keyword `fmod` followed by the module's name, and ends with the keyword `endfm`. Such modules can be parameterized, as in the following module for list operations:

```
fmod LIST[X :: TRIV] is
  protecting NAT BOOL .
  sort List .
  subsort Elt < List .
  op __ : List List -> List [assoc id: nil] .
  op length : List -> Nat .
  op _in_ : Elt List -> Bool .
  vars E E' : Elt .
  var L : List .
  eq length(nil) = 0 .
  eq length(E L) = 1 + length(L) .
  eq E in nil = false .
  eq E in (E' L) = if E == E' then true
       else E in L fi .
endfm
```

In parameterized modules, the properties that the parameter must satisfy are specified by one or more *parameter theories*. In this case, the (functional) parameter theory is the trivial theory `TRIV`

```
fth TRIV is
  sort Elt .
endft
```

which only requires a set `Elt` of elements. Such a parameterized module can then be instantiated by providing an interpretation mapping the parameter sort `Elt` to a sort in the module chosen as the actual parameter. For example, if we interpret `Elt` as the sort `Nat` in the `NAT` module, then we can instantiate this module to form lists of natural numbers by writing

```
make NAT-LIST is LIST[Nat] endmk
```

A module contains sort and subsort declarations introduced by the keywords `sort(s)` and `subsort(s)` stating the different sorts of data manipulated by the module and how those sorts are related. As in OBJ3 [20], MaudeLog's type structure is *order-sorted* [18]; therefore, it is possible to declare one sort as a *subsort* of another; for example, the declaration `Elt < List` states that every data element is a list (of length one). It is also possible to *overload* function symbols for operations that are defined at several levels of a sort hierarchy and agree on their results when restricted to common subsorts; for example, an addition operation `_+_` may be defined for sorts `Nat`, `Int`, and `Rat` of natural, integer, and rational numbers with

```
Nat < Int < Rat .
```

Each function provided by the module, as well as the sorts of its arguments and the sort of its result, is introduced using the keyword `op`. The syntax is user-definable, and, in addition to standard parenthesized notation (`length`), permits specifying function symbols in "prefix" (e.g., a successor function `s_` for natural numbers), "infix" (`_in_`), or any "mixfix" (e.g., a remove function `remove_from_`) combinations, including "empty syntax" (`__`). Note that the function `__` has been declared *associative* and has the constant `nil` as its *identity* element.

Variables to be used for defining equations are declared with their corresponding sorts, and then equations are given; such equations provide the actual "code" of the module. Deduction with such equations is a typed variant of equational logic called *order-sorted equational logic*. However, operationally, only deduction from left to right by rewriting is performed, as explained below. As in OBJ3, the *mathematical semantics* of a MaudeLog functional module is the *initial* order-sorted algebra [18] satisfying the equations declared in the module (more on this in Section 3.4).

The statement `protecting NAT BOOL` imports `NAT` and `BOOL` as *submodules* of `LIST` and asserts that neither the natural numbers nor the Booleans are modified in the sense that no new data of sorts `Nat` or `Bool` are added, and different numbers or different truth values are not identified by the new equations declared in `LIST`.

To compute with a functional module, one performs equational simplification by using the equations from left to right until no more simplifications are possible. Note that this can be done *concurrently*, i.e., applying several equations at once, a style of computation that we call *concurrent rewriting*. Therefore, functional modules—and, as we shall see later, object-oriented modules—are intrinsically parallel.

In MaudeLog, the rules in a functional module are always assumed to be Church-Rosser (see [22, 15] for further background on the subject, and [24] for corresponding order-sorted versions of such notions), but as we shall see later this is not assumed for object-oriented modules, which do *not* have a functional interpretation.

2.1.2 Object-Oriented Modules

Object-oriented modules define object-oriented databases as concurrent systems of objects that communicate with each other by means of messages. An *object* in a given state is represented as a term

$$\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$$

where O is the object's name or identifier, C is its class, the a_i's are the names of the object's *attribute identifiers*, and the v_i's are the corresponding *values*.

The *configuration* is the distributed state of an object-oriented database and is represented as a multiset of objects and messages according to the following syntax:

```
subsorts Object Message < Configuration .
op __ : Configuration Configuration
  -> Configuration [assoc comm id: null] .
```

where the function __ is declared to be associative and commutative with identity null and is interpreted as multiset union, and where the sorts Object and Message are subsorts of Configuration and generate data of that sort by multiset union.

MaudeLog's syntax for object-oriented modules is illustrated by the object-oriented module ACCNT below which specifies the dynamic behavior of objects in a very simple class Accnt of bank accounts, each having a bal(ance) attribute, which may receive messages for crediting or debiting the account, or for transferring funds between two accounts. We assume an already given functional module REAL for real numbers with a subsort relation NNReal < Real corresponding to the inclusion of the nonnegative reals (i.e., reals greater or equal than zero) into the reals, and with an ordering predicate _>=_

```
omod ACCNT is
  protecting REAL .
  class Accnt | bal: NNReal .
  msgs credit debit : OId NNReal -> Msg .
  msg transfer_from_to_ : NNReal OId OId -> Msg .
  vars A B : OId .
  vars M N N' : NNReal .
  rl credit(A,M) < A : Accnt | bal: N > =>
    < A : Accnt | bal: N + M > .
  rl debit(A,M) < A : Accnt | bal: N > =>
    < A : Accnt | bal: N - M > if N >= M .
  rl transfer M from A to B
    < A : Accnt | bal: N > < B : Accnt | bal: N' >
    => < A : Accnt | bal: N - M >
       < B : Accnt | bal: N' + M > if N >= M .
endom
```

After the keyword class, the name of the class—in this case Accnt—is given, followed by a "|" and by a list of pairs of the form a: S separated by commas, where a is an attribute identifier and S is the sort inside which the values of such an attribute identifier must range in the given class. In this example, the only attribute of an account is its bal(ance), which is declared to be a value in NNReal. The three kinds of messages involving accounts are credit, debit, and transfer messages, whose user definable syntax is introduced by the keyword msg. The rewrite rules (introduced by the keyword rl) specify in a declarative way the dynamic behavior associated with the credit, debit, and transfer messages. Note that the existence of a *configuration* having the structure of a multiset of objects and messages is implicitly assumed in the definition of any object-oriented module; in fact, the multiset union operator appears, with empty syntax, in each of the above rewrite rules.

In general, an object-oriented module contains class and subclass declarations introduced by the keywords class(es) and subclass(es) declaring the different classes of objects manipulated by the module and how those classes are related. For example, we can define an object-oriented module CHK-ACCNT of checking accounts extending the ACCNT module by introducing a subclass ChkAccnt of Accnt with a new attribute chk-hist recording the history of checks cashed in the account.

```
omod CHK-ACCNT is
  extending ACCNT .
  protecting
    LIST[2TUPLE[Nat,NNReal]]*(sort List to ChkHist) .
  class ChkAccnt | chk-hist: ChkHist .
  subclass ChkAccnt < Accnt .
  msg chk_#_amt_ : OId Nat NNReal -> Msg .
  var A : OId .
  vars M N : NNReal .
  var K : Nat .
  var H : ChkHist .
  rl (chk A # K amt M)
    < A : ChkAccnt | bal: N, chk-hist: H >
    => < A : ChkAccnt | bal: N - M,
      chk-hist: H << K ; M >> > if N >= M .
endom
```

The protecting statement imports a data type of lists of 2-tuples (pairs denoted <<_;_>>) consisting of a natural number and a nonnegative real, and renames the principal sort List to ChkHist. The checking history of the account is then represented as a list of such pairs with the first number in the pair corresponding to the check number, and the second number corresponding to the check's amount.

We finally note that, as in the case of functional modules, object-oriented modules can be parameterized by parameter theories. For examples of parameterized object-oriented modules we refer the reader to [29].

2.2 Updates and Queries

In MaudeLog, an object-oriented database evolves by active objects manipulating attributes and exchanging messages. Intuitively, we can think of messages as traveling to come into contact with the objects to which they are sent and then either causing state change or querying the state of an object.

For example, Figure 1 provides a snapshot in the evolution of a simple configuration of bank accounts. The state before the update consists of three objects and five messages. The state change consists of executing three of the messages on the objects to which they are sent, leading to a state consisting of three objects and two messages.

The evolution is captured by concurrent rewriting (modulo *ACI*—Associativity, Commutativity, and Identity) of the configuration by means of rewrite rules specific to each particular database, whose left-hand and right-hand sides may in general involve patterns for several objects and messages. By specializing to patterns involving only one object and one message in their left-hand side, we can obtain an abstract and truly concurrent version of the Actor model [5, 6] (see [27]).

Implicit in an object-oriented module definition is the capacity to query the attributes of an object by means of messages. In the ACCNT example, an object O can query the balance of account A by means of the message

```
A . bal query Q replyto O
```

where Q is a query identification number. If A has balance N at the time of answering the message, then O will get back the message

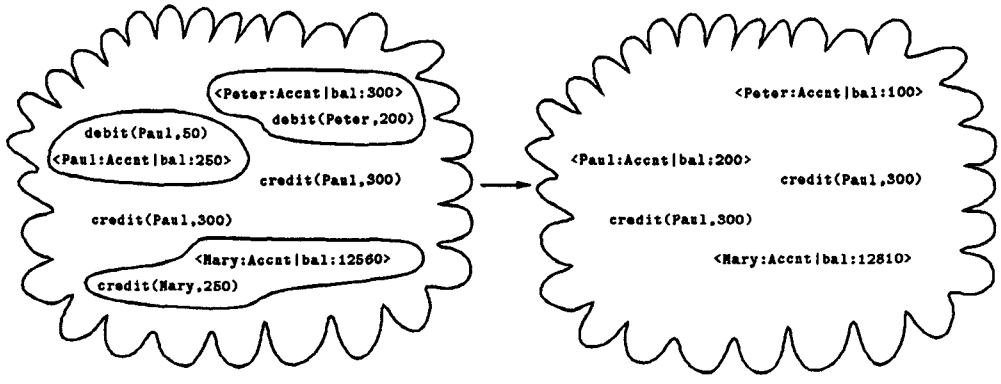

Figure 1: Concurrent rewriting of bank accounts.

```
to O ans-to Q : A . bal is N
```

The rewrite rule implicit in the module to handle such queries and answers is in this case

```
rl (A . bal query Q replyto O)
  < A : Accnt | bal: N > =>
  < A : Accnt | bal: N >
  (to O ans-to Q : A . bal is N)
```

More complex and perhaps parameterized queries that extract information about the entire state of an object can be handled in a similar way by introducing the appropriate messages and rewrite rules explicitly. Such messages are sometimes called *derived* or *computed* attributes, and other times are referred to as invocations of *methods*[3]. For example, the amount of interest accrued by an interest-yielding checking account can be viewed as a computed attribute that depends on the current balance and the previous financial history of the account, and that has as a parameter the time period over which the accrual is computed.

In MaudeLog it is also possible to ask queries involving logical variables and requiring a possibly complex inspection of the entire database state, i.e., of the configuration. For example, the query

```
all A : Accnt | (A . bal) >= 500 .
```

should be answered by providing the set of all account identifiers that have at present a balance greater than or equal to \$500. We discuss queries with logical variables further in Section 4.1.

3 Rewriting Logic

This section gives a brief introduction to rewriting logic, which is the semantic basis of MaudeLog in the sense that a MaudeLog module is, except for some syntactic sugar, a theory in rewriting logic. Concurrent computation by rewriting then exactly corresponds to logical deduction. In fact, rewriting logic is a logic of change that allows correct reasoning about the evolution of concurrent or distributed systems.

In addition, the initial model semantics of MaudeLog's modules is briefly discussed. In the context of the logical foundations of databases, the importance of an initial model semantics has been recently stressed by Beeri [9], and by Beeri and Milo [10].

For functional modules, the initial models are indeed initial algebras so that we have an order-sorted version of the well-known initial algebra semantics [19, 18] of equational logic. What is in fact surprising is that we also have an initial model semantics—based on a different model theory, namely in the models for rewriting logic—for object-oriented modules which includes all the dynamic aspects of such modules. Although viewed as very desirable, the possibility of an initial model semantics for object-oriented databases seems to have been considered most challenging by researchers because of their dynamic and state-oriented character (see for example [9, 10]).

As already mentioned, MaudeLog's type structure for both functional and object-oriented modules is order-sorted [18], and therefore supports subtypes (called subsorts), subclasses, and overloading. However, to simplify the exposition we present an unsorted version of rewriting logic.

3.1 Basic Universal Algebra

A set Σ of function symbols is a ranked alphabet $\Sigma = \{\Sigma_n \mid n \in \mathbb{N}\}$. A Σ-algebra is then a set A together with an assignment of a function $f_A : A^n \longrightarrow A$ for each $f \in \Sigma_n$ with $n \in \mathbb{N}$. We denote by T_Σ the Σ-algebra of ground Σ-terms, and by $T_\Sigma(X)$ the Σ-algebra of Σ-terms with variables in a set X. Similarly, given a set E of Σ-equations, $T_{\Sigma,E}$ denotes the Σ-algebra of equivalence classes of ground Σ-terms modulo the equations E (i.e., modulo provable equality using the equations E); in the same way, $T_{\Sigma,E}(X)$ denotes the Σ-algebra of equivalence classes of Σ-terms with variables in X modulo the equations E. Let $[t]_E$ or just $[t]$ denote the E-equivalence class of t.

Given a term $t \in T_\Sigma(\{x_1, \ldots, x_n\})$, and terms $u_1, \ldots, u_n$, $t(u_1/x_1, \ldots, u_n/x_n)$ denotes the term obtained from t by *simultaneously substituting* u_i for x_i, $i = 1, \ldots, n$. To simplify notation, we denote a sequence $a_1, \ldots, a_n$ by $\overline{a}$. With this notation, $t(u_1/x_1, \ldots, u_n/x_n)$ can be abbreviated to $t(\overline{u}/\overline{x})$.

[3] As already mentioned, we prefer to speak of *messages*. Note that in our approach some messages, such as the `credit` and `debit` messages, produce state updates, whereas other messages query objects and extract relevant information about their basic or computed attributes without changing their state.

3.2 The Rules of Rewriting Logic

We are now ready to introduce rewriting logic. A *signature* in this logic is a pair (Σ, E) with Σ a ranked alphabet of function symbols and E a set of Σ-equations. Rewriting will operate on equivalence classes of terms modulo the equations E. In this way, we free rewriting from the syntactic constraints of a term representation and gain a much greater flexibility in deciding what counts as a *data structure*; for example, string rewriting is obtained by imposing associativity, and multiset rewriting by imposing associativity and commutativity. Of course, standard term rewriting is obtained as the particular case in which the set E of equations is empty. The idea of rewriting in equivalence classes is well known (see for example [22, 15]).

Given a signature (Σ, E), *sentences* of the logic are sequents of the form $[t]_E \longrightarrow [t']_E$, where t, t' are Σ-terms that may involve some variables from the countably infinite set $X = \{x_1, \ldots, x_n, \ldots\}$. A *theory* in this logic, called a *rewrite theory*, is a slight generalization of the usual notion of theory—which is typically defined as a pair consisting of a signature and a set of sentences for it—in that, in addition, we allow sentences to be labeled.

Definition 1 A (*labeled*) *rewrite theory* $\mathcal{R}$ is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where Σ is a ranked alphabet of function symbols, E is a set of Σ-equations, L is a set of *labels*, and R is a set of pairs $R \subseteq L \times (T_{\Sigma,E}(X)^2)$ whose first component is a label and whose second component is a pair of E-equivalence classes of terms, with $X = \{x_1, \ldots, x_n, \ldots\}$ a countably infinite set of variables. Elements of R are called *rewrite rules*[4]. We understand a rule $(r, ([t], [t']))$ as a labeled sequent and use for it the notation $r : [t] \longrightarrow [t']$. To indicate that $\{x_1, \ldots, x_n\}$ is the set of variables occurring in either t or t', we write $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$, or in abbreviated notation $r : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$. □

Given a rewrite theory $\mathcal{R}$, we say that $\mathcal{R}$ *entails* a sequent $[t] \longrightarrow [t']$ and write $\mathcal{R} \vdash [t] \longrightarrow [t']$ if and only if $[t] \longrightarrow [t']$ can be obtained by finite applications of the following *rules of deduction*:

1. **Reflexivity.** For each $[t] \in T_{\Sigma,E}(X)$,

$$\frac{}{[t] \longrightarrow [t]}$$

2. **Congruence.** For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{[t_1] \longrightarrow [t'_1] \quad \ldots \quad [t_n] \longrightarrow [t'_n]}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]}$$

3. **Replacement.** For each rewrite rule $r : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ in R,

$$\frac{[w_1] \longrightarrow [w'_1] \quad \ldots \quad [w_n] \longrightarrow [w'_n]}{[t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}$$

4. **Transitivity.**

$$\frac{[t_1] \longrightarrow [t_2] \quad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}$$

Equational logic (modulo a set E of axioms) is obtained from rewriting logic by adding the following rule:

5. **Symmetry.**

$$\frac{[t_1] \longrightarrow [t_2]}{[t_2] \longrightarrow [t_1]}$$

With this new rule, sequents derivable in equational logic are *bidirectional*; therefore we can adopt the notation $[t] \leftrightarrow [t']$ throughout and call such bidirectional sequents *equations*.

A nice consequence of having defined rewriting logic is that concurrent rewriting, rather than emerging as an operational notion, actually *coincides* with deduction in such a logic.

Definition 2 Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, a (Σ, E)-sequent $[t] \longrightarrow [t']$ is called a *concurrent $\mathcal{R}$-rewrite* (or just a *rewrite*) iff it can be derived from $\mathcal{R}$ by finite applications of the rules 1–4. □

For example, the concurrent update of bank accounts in Figure 1 is a concurrent rewrite for the rewrite theory of which the `ACCNT` module is a sugared version. Note that in this example the equations E consist of the associativity, commutativity, and identity axioms for the multiset union operator used to form configurations, which is implicit in the specification of object-oriented modules.

3.3 The Meaning of Rewriting Logic

A logic worth its salt should be understood as a method of correct reasoning about some class of entities, not as an empty formal game. For equational logic, the entities in question are sets, functions between them, and the relation of identity between elements. For rewriting logic, the entities in question are *concurrent systems* having *states*, and evolving by means of *transitions*. The signature of a rewrite theory describes a particular structure for the states of a system—e.g., multiset, binary tree, etc.—so that the states can be distributed according to such a structure. The rewrite rules in the theory describe which *elementary* concurrent transitions are possible. What the rules of rewriting logic allow us to reason correctly about is which *general* concurrent transitions are possible in a system satisfying such a description. Clearly, concurrent systems should be the *models* giving a semantic interpretation to rewriting logic, in the same way that algebras are the models giving a semantic interpretation to equational logic. A precise account of the model theory of rewriting logic, giving rise to the initial model semantics for MaudeLog modules that is fully consistent with the above system-oriented interpretation, is given in [27]; Section 3.4 below gives a brief, informal, explanation of such semantics.

Therefore, in rewriting logic a sequent $[t] \longrightarrow [t']$ should not be read as "$[t]$ *equals* $[t']$," but as "$[t]$ *becomes* $[t']$." Clearly, rewriting logic is a logic of *becoming* or *change*, not a logic of equality in a static Platonic sense. The apparently innocent step of adding the symmetry rule is in fact a

[4] To simplify the exposition the logic is given for the case of *unconditional* rewrite rules. However, all the ideas and results presented here have been extended in [27] to conditional rewrite rules of the general form

$$r : [t] \longrightarrow [t'] \text{ if } [u_1] \longrightarrow [v_1] \wedge \ldots \wedge [u_k] \longrightarrow [v_k].$$

This of course increases considerably the expressive power of rewrite theories.

very strong restriction, namely assuming that *all change is reversible*, thus bringing us into a timeless Platonic realm in which "before" and "after" have been identified.

A related observation is that $[t]$ should not be understood as a *term* in the usual first-order logic sense, but as a *proposition*—built up using the *propositional connectives* in Σ—that asserts being in a certain *state* having a certain *structure*. However, unlike most other logics, the logical connectives Σ and their structural properties E are entirely *user-definable*. This provides great flexibility for considering many different state structures and makes rewriting logic very general in its capacity to deal with many different types of concurrent systems. This generality is discused at length in [27].

In summary, the rules of rewriting logic are rules to reason about *change in a concurrent system*. They allow us to draw valid conclusions about the evolution of the system from certain basic types of change known to be possible. This is summarized below.

State	$\leftrightarrow$ *Term*	$\leftrightarrow$ *Proposition*
Transition	$\leftrightarrow$ *Rewriting*	$\leftrightarrow$ *Deduction*
Dist. Struct.	$\leftrightarrow$ *Alg. Struct.*	$\leftrightarrow$ *Prop. Conn.*

3.4 Initial Model Semantics

In MaudeLog, modules are logical theories. A functional module is a theory in order-sorted equational logic, and an object-oriented module is (a sugared version of) a theory in rewriting logic. Since there are two different logics, there are also two different initial model semantics that are however systematically related by a map of logics.

In the case of functional modules, their semantics is the initial algebra of their corresponding equational theory. Such an equational theory consists of an order-sorted signature and a set of equations. The signature consists of a set of sorts partially ordered by a subsort relation and a set of function symbols, where each function symbol may be overloaded so that several assignments of the form $f : s_1 \dots s_n \longrightarrow s$ are possible for the same f. The equations are pairs of well-formed terms separated by an equality sign (or more generally conditional equations) and are implicitly universally quantified by the relevant variables, each with a given sort. Some syntactic restrictions that need not concern us now apply to signatures and to equations [18].

A model for such an order-sorted theory is called an *order-sorted algebra* [18] and consists of a set A_s for each sort symbol s, so that if $s \leq s'$ then $A_s \subseteq A_{s'}$, together with a function $f_A : A_{s_1} \times \dots \times A_{s_n} \longrightarrow A_s$ for each function specification $f : s_1 \dots s_n \longrightarrow s$ in such a way that if another function specification $f : s'_1 \dots s'_n \longrightarrow s'$ has been given, with $s'_i \leq s_i$, $i = 1, \dots, n$, and $s' \leq s$, then the function $f_A : A_{s'_1} \times \dots \times A_{s'_n} \longrightarrow A_{s'}$ is just the restriction of the function $f_A : A_{s_1} \times \dots \times A_{s_n} \longrightarrow A_s$ to the subset $A_{s'_1} \times \dots \times A_{s'_n}$. In addition, the algebra A is required to satisfy the equations E of the theory. The number hierarchy from the naturals to the rationals (and of course beyond) provides a good example of an order-sorted algebra.

The mathematical semantics of a functional module is the *initial order-sorted algebra* satisfying the equations declared in the module. Abstractly, such an algebra can be characterized up to isomorphism as an algebra that satisfies the equations and from which there is a unique homomorphism to any other algebra satisfying the equations. Concretely, such an algebra can be built out of the proof theory of order-sorted equational logic as the quotient of the free algebra of ground terms under the equivalence relation of provable equality [18].

In a similar way, but based on a different logic and model theory, the semantics of an object-oriented module is given by the initial model of the order-sorted rewrite theory that it describes. The signature (Σ, E) of such a rewrite theory is itself an order-sorted equational theory specifying both the type structure and the structural axioms, and the rewrite rules are implicitly universally quantified by their sorted variables.

The models of such a rewrite theory are, informally speaking, concurrent systems whose states may have different sorts according to the sort structure of the theory, and whose concurrent transitions are governed by the rewrite rules in the theory. Mathematically, the concurrent states form an order-sorted algebra, and the concurrent transitions have also an algebraic structure given by the deduction rules for order-sorted rewriting logic. In particular, the reflexivity rule ensures the presence of *idle* or *identity* transitions, and the transitivity rule provides a *composition* operator for pairs of transitions sharing a common intermediate state. Since composition of transitions is assumed associative, transitions give rise to a *category* structure [25] for each of the sorts, in such a way that the functions of the signature become *functors*, and the rewrite rules become *natural transformations*.

Complete details about the model theory of rewriting logic (in the unsorted case), including soundness, completeness, and initiality theorems can be found in [27]. The key point for our present purposes is that rewrite theories have initial models; they can be built out of the proof theory of rewriting logic as concurrent systems having as states equivalence classes of ground terms modulo the structural axioms E, and whose transitions are equivalence classes of proof expressions modulo appropriate axioms imposing the categorical and algebraic structure on the transitions [27]. Given the identification of proofs with concurrent computations, such equivalence classes of proof expressions provide an abstract, equational, notion of *true concurrency*, so that each of the equivalent proof expressions is a different syntactic description of the same concurrent computation.

We finish this section by remarking that, although different, the two logics involved as well as their corresponding model theories, namely order-sorted equational logic and order-sorted rewriting logic, are systematically related by an embedding of logics

$$OSEqtl \longrightarrow OSRWLogic.$$

The details of this embedding are discussed in [28]. At the language level, such a map corresponds to the inclusion of MaudeLog's functional modules within the language and provides a simple and rigorous declarative unification of the functional and the concurrent object-oriented paradigms.

4 A Semantics for OODB

Now that MaudeLog's semantic basis in rewriting logic has been introduced, we explain in more detail how basic concepts such as schemas, databases and database states, updates, and queries can be formally expressed in such a framework. In the same vein, we also discuss inheritance issues and their semantics, insisting on a sharp separation between classification of data into taxonomic hierarchies by means of *class inheritance*, and reuse and evolution of modules by means of *module inheritance*.

4.1 Schemas and Databases

A *schema* is a rewrite theory, the rules of which specify the dynamic behavior of an object-oriented database. A *database* over the schema is the initial model of the rewrite theory, which represents a concurrent system of active objects. A database *state* is a configuration, which evolves by concurrent rewriting using rules of the schema. Dynamic evolution exactly corresponds to *deduction* in rewriting logic. In other words, the states S that are *reachable* from an initial state S_0 are exactly those such that the sequent $S_0 \longrightarrow S$ is *provable* in rewriting logic using rules of the schema.

Messages serve two main purposes, namely state change and communication. Database *updates* are produced by messages that change the state of an object according to appropriate rewrite rules. Messages that do not change the state of the objects in the database but only request or communicate information support query processing of attributes, and of derived or computed attributes, where such computed attributes can have parameters.

In addition to queries about the basic or computed attributes of a single object, more complex queries extracting information about an entire database state are also supported by MaudeLog. In particular, queries involving logical variables such as the example

```
all A : Accnt | (A . bal) >= 500 .
```

already discussed in Section 2.2 are supported. Such queries are sugared versions of existential formulas of the form

$$\exists \overline{x} \;\; [u_1(\overline{x})] \longrightarrow [v_1(\overline{x})] \wedge \ldots \wedge [u_k(\overline{x})] \longrightarrow [v_k(\overline{x})]$$

and their answers correspond to proofs or "witnesses" of such existential formulas in the rewrite theory specified by the schema. For the above example, the de-sugared version is the formula

```
(∃A : OId)      (< A : Accnt | bal: N > in C)
    ⟶ true
∧ (N >= 500)  ⟶ true.
```

where C is the current database state, and all the answers correspond to the different ground substitutions of A that prove such a formula.

In connection with existential queries it is worth pointing out that rewriting logic generalizes Horn logic in the sense that there is an embedding of logics

$$OSHorn \longrightarrow OSRWLogic$$

that systematically relates order-sorted Horn logic to order-sorted rewriting logic. The details of this map are discussed in [28]. In particular, recursive queries with logical variables in the Datalog style can be handled within the same formal framework. Note also that, given MaudeLog's type structure, the unification performed on logical variables is order-sorted unification [30]. For objects this yields a variant of feature unification [32, 23, 13] the details of which we are currently investigating.

An important topic that we hope to address in future work is the tradeoff between message passing and instantiation of logical variables as computational mechanisms for query processing. This is because in MaudeLog messages can not only be sent from one object to another; they can also be broadcast to all the objects in a class [29]. For example, to find out how many accounts have a balance above \$500, an appropriate message could be broadcast to all the accounts in the database, with only those having a positive answer responding back with their object identifier.

4.2 Inheritance

In MaudeLog, inheritance is supported at two different levels that are carefully distinguished. At a type-theoretic level of data sorts and object classes, sort and class inheritance provides a means of classifying data and objects into taxonomic hierarchies. At the level of modules, module inheritance supports modularity, reuse, and ease of evolution by arranging modules into hierarchies and by providing a rich algebra of module composition operations.

4.2.1 Sort and Class Inheritance

As already mentioned, MaudeLog's type structure for functional as well as for object-oriented modules is *order-sorted*. This means that the data elements of functional modules are classified into sort hierarchies in such a way that if $s \leq s'$ holds between sorts, then $A_s \subseteq A_{s'}$ holds between the corresponding sets of data elements in the initial model. In addition, overloading is supported, meaning that the behavior of a function on subsorts is entirely determined by its behavior on supersorts.

Since rewriting logic is order-sorted, class inheritance is also directly supported by MaudeLog's type structure. A subclass declaration `C < C'` is just a special case of a subsort declaration `C < C'`, and guarantees that any object in a subclass is also an object in a superclass.

Specifically, the effect of a subclass declaration is that the attributes, messages and rules of all the superclasses as well as the newly defined attributes, messages and rules of the subclass characterize the structure and behavior of the objects in the subclass [29].

4.2.2 Module Inheritance

There are cases in which one wants to *modify* the original code to adapt it to a different situation. The class inheritance mechanisms will *not* help in such cases. Rather than violating class inheritance to force upon them the job of modifying code, the solution adopted in MaudeLog is to insist on keeping the order-sorted semantics for class inheritance, and to provide module inheritance mechanisms to do the job of code modification. This distinction between

the level of classes (more generally sorts) and the level of modules was already clearly made in the FOOPS language (besides the original paper [17], see also [21] for a very good discussion of inheritance issues and of the class-module distinction in the context of FOOPS), and indeed goes back to the distinction between sorts and modules in OBJ [20].

In MaudeLog, code in modules can be modified or adapted for new purposes by means of a variety of *module operations*—and combinations of several such operations in *module expressions*—whose overall effect is to provide a very flexible style of software reuse that can be summarized under the name of *module inheritance*. Module operations of this kind include:

1. *importing* a module in a `protecting`, `extending`, or `using` mode;
2. *adding new equations or rules* to an imported module;
3. *renaming* some of the sorts or operations of a module;
4. *instantiating* a parameterized module;
5. *adding modules* to form their union;
6. *redefining* a function so that its sort and syntax requirements are kept intact, but its semantics can be changed by discarding previously given equations or rules involving the function so that new equations or rules can then be given in their place;
7. *removing* a sort or function altogether along with the equations or rules that depend on it so that it can be either discarded or replaced by another sort or function with different syntax and semantics.

The operations 1–5 are exactly as in OBJ3 [20]. The operations 6–7 are new and give a simple solution to the thorny problem of *message specialization* without complicating the class inheritance relation, which remains based on the order-sorted semantics. The need for message specialization, i.e., for providing a different behavior for a message, arises frequently in practice. For example, a bank may at some point want to introduce a new kind of checking accounts in which there is a charge of 50 cents for each cashed check. The updating of an account's balance upon receipt of a message of type `(chk A # K amt M)` has to be modified by the extra 50 cents charge. The problem is that if the new class of checking accounts with checking charges is defined as a subclass of the old, then the nice property supported by class inheritance of inheriting the rules from superclasses mentioned in Section 4.2.1 is destroyed, because the rules from the superclass should *not* be inherited in the new subclass and would in fact produce the wrong behavior.

Our solution is to understand it as a module inheritance problem, and to carefully distinguish it from class inheritance. In this case, it is the *modules* in which the classes are defined that stand in an inheritance relation, not the classes themselves. In particular, the *redefine* operation, with keyword `rdfn`, provides the appropriate way of modifying and inheriting the `CHK-ACCNT` module. A detailed discussion of this example illustrating the differences between class and module inheritance can be found in [29].

In real life, databases are always in constant change. Not only the data but also the very structure of the database are always evolving, in the sense that new applications always demand new ways of structuring or extending the data. Flexible, highly modular, and parameterized approaches to the evolution of schemas are of great practical importance, especially for large databases. MaudeLog's class and module inheritance mechanisms provide strong support for schema evolution. Class inheritance supports further refinement of class hierarchies in a way consistent with the behavior of previously defined superclasses, whereas module inheritance supports a highly modular and parameterized way of evolving and adapting previously defined modules and classes to new modules with different behavior.

5 Concluding Remarks

We have proposed a semantic foundation for object-oriented databases based on rewriting logic, and a declarative language based on that logic called MaudeLog that integrates query, update, and programming aspects. We have discussed some of the key advantages of our proposed solution, including its simple logical nature, the fact that dynamic aspects are directly addressed, the integration of user-definable algebraic data types within the framework, and the existence of initial models.

Much work remains ahead. In particular, the query language aspects need to be investigated in much more detail, including deduction, support for feature unification when querying objects, and the appropriate balance between message passing and unification mechanisms in query answering.

Similarly, views need to be studied in detail, and the connections between our approach based on theory interpretations and approaches based on an understanding of views as queries such as that of Abiteboul and Bonner [1] should be investigated.

Finally, implementation aspects should be studied, combining the flexibility of machine-independence with the capacity for supporting interface modules written in conventional languages [31], and supporting the linkage with heterogeneous databases that would permit using MaudeLog as a very high level *mediator language* [33, 34].

Acknowledgements

We thank Narciso Martí-Oliet for his very helpful comments and for his careful reading of the manuscript.

References

[1] S. Abiteboul and A. Bonner. Objects and views. In *Proc. ACM SIGMOD*, pages 238–247, 1991.

[2] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD*, pages 159–173, 1989.

[3] S. Abiteboul and P. Kanellakis. The two facets of object-oriented data models. In *IEEE Data Engineering Bulletin* 14(2):3–7, 1991.

[4] S. Abiteboul, P. Kanellakis, and E. Waller. Method schemas. In *Proc. 9th PODS*, pages 16–27. ACM, 1990.

[5] G. Agha. *Actors*. MIT Press, 1986.

[6] G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro (editors), *Object-Oriented Concurrent Programming*, pages 37–53. MIT Press, 1988.

[7] M. Atkinson, P. Richard, and P. Trinder. Bulk types for large scale programming. In *Proc. Next Generation Information System Technology*, pages 228–250, Springer LNCS 504, 1991.

[8] C. Beeri. Theoretical foundations for oodb's – a personal perspective. In *IEEE Data Engineering Bulletin* 14(2):8–12, 1991.

[9] C. Beeri. New data models and languages—the challenge. In *Proc. 11th PODS*, pages 1–15. ACM, 1992.

[10] C. Beeri and T. Milo. Functional and predicative programming in oodb's. In *Proc. 11th PODS*, pages 176–190. ACM, 1992.

[11] E. Bertino. A view mechanism for object-oriented databases. In *Proc. EDBT*, pages 136–151, Springer LNCS 580, 1992.

[12] V. Breazu-Tannen and R. Subrahmanyam. Logical and computational aspects of programming with sets/bags/lists. In *Proc. ICALP*, pages 60–75, Springer LNCS 510, 1991.

[13] B. Carpenter. *The Logic of Typed Feature Structures*. Cambridge University Press, 1992.

[14] W. Chen, M. Kifer, and D.S. Warren. Hilog as a platform for database languages (or why predicate calculus is not enough). In *Proc. 2nd Int'l Workshop on Database Programming Languages*, pages 315–329, 1989.

[15] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen (editor), *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. North-Holland, 1990.

[16] J. Goguen. Merged views, closed worlds, and ordered sorts: Some novel database features in OBJ. In A. Borgida and P. Buneman (editors), *Proc. 1982 Workshop on Database Interfaces*, pages 38–47, University of Pennsylvania, Computer Science Department, 1985.

[17] J. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner (editors), *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.

[18] J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science* 105:217–273, 1992.

[19] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology, IV*, R. Yeh (editor), Prentice-Hall, 1978, pages 80-149.

[20] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. To appear in J. Goguen (editor), *Applications of Algebraic Specification Using OBJ*, Cambridge University Press.

[21] J. Goguen and D. Wolfram. On types and FOOPS. In *Proc. IFIP Working Group 2.6 Working Conference on Database Semantics: Object-Oriented Databases: Analysis, Design and Construction, 1990*.

[22] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the Association for Computing Machinery*, 27:797–821, 1980.

[23] M. Kifer and G. Lausen. F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *Proc. ACM SIGMOD*, pages 134–146, 1989.

[24] C. Kirchner, H. Kirchner, and J. Meseguer. Operational semantics of OBJ3. In T. Lepistö and A. Salomaa (editors), *Proc. 15th Intl. Coll. on Automata, Languages and Programming*, July 1988, pages 287–301, Springer LNCS 317, 1988.

[25] S. MacLane. *Categories for the working mathematician*. Springer-Verlag, 1971.

[26] D. Maier. A logic for objects. In *Proc. Workshop on Foundations of Deductive Databases and Logic Programming, Washington, D.C.*, pages 6–26, 1986.

[27] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science* 96(1):73–155, 1992.

[28] J. Meseguer. Multiparadigm logic programming. In H. Kirchner and G. Levi (editors), *Proc. 3rd Intl. Conf. on Algebraic and Logic Programming*, pages 158–200, Springer LNCS 632, 1992.

[29] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. To appear in G. Agha, P. Wegner, and A. Yonezawa (editors), *Research Directions in Object-Based Concurrency*, MIT Press, 1993.

[30] J. Meseguer, J. Goguen, and G. Smolka. Order-sorted unification. *J. Symbolic Computation* 8:383–413, 1989.

[31] J. Meseguer and T. Winkler. Parallel programming in Maude. In J.-P. Banâtre and D. Le Mètayer (editors), *Research Directions in High-level Parallel Programming Languages*, pages 253–293, Springer LNCS 574, 1992.

[32] G. Smolka and H. Aït-Kaci. Inheritance hierarchies: Semantics and unification. *J. Symbolic Computation*, 7:343–370, 1989.

[33] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer* 25(3):38–49, March 1992.

[34] G. Wiederhold, P. Wegner, and S. Ceri. Toward megaprogramming. *CACM* 35(11):89–99, November 1992.