Chimera: A Model and Language for Active DOOD Systems*

Stefano Ceri
Dip. di Elettronica e Informazione
Politecnico di Milano
Piazza L. Da Vinci - I-20133 Milano
Italy

Rainer Manthey
Department of Computer Science
University of Bonn
Römerstrasse 164, D-53117 Bonn
Germany

Abstract
Chimera is a novel database model and language which has been designed as a joint conceptual interface of the IDEA project, a major European cooperation initiative aiming at the integration of object-oriented active and deductive database technology. In this paper, we present a view of the main features of Chimera and discuss the design choices made. The most remarkable characteristic of Chimera is the fact that fully developed rule languages for both active and deductive rules have been integrated in an object-oriented context. Rules are in the center of interest of the IDEA project, which aims at developing prototypical components of a future “intelligent” DBMS.

1 Introduction

In this paper, we introduce a major attempt towards integrating active, deductive and object-oriented data modeling and manipulation techniques within a unified framework. The resulting model and language, called Chimera, has been designed as a joint conceptual interface for our work within the IDEA project, a four-years activity sponsored by the ESPRIT program of the Commission of the European Community. The project consortium involves partners from Belgium, France, Germany, Italy and Spain. The overall aim of the IDEA project has been the design and prototyping of an “intelligent database system”, offering a conceptual interface that supports a variety of novel features proposed and developed recently within the DB research community: an object-oriented data model, deductive rules (for the definition of derived concepts), static integrity constraints (for the definition of legal DB states) and active rules (for the definition of reactive behaviour of the system in addition to the methods of the data model). Systems integrating deductive and object-oriented functionality have been called “DOOD systems” for some time now. Active DOOD systems in addition provide the concept of an active rule, thus naturally extending representation power and operational strength.

Chimera is the name of a famous mythical creature composed of a lion, a goat and a serpent. When designing Chimera, we often felt like creating such

*This work was supported in part by the Commission of the European Community under ESPRIT project number 6533 (IDEA)
a monster indeed! On the one hand, the integration of concepts from three of the most active and productive areas of current research caused a lot of problems, because there are so many possible choices and design parameters to be considered. On the other hand, doing such an exercise within the context of a European research consortium composed of a very heterogeneous set of partners - from different nations, from industry and academia, with very different background and interests - is a difficult and exhausting affair. However, we feel that this effort has been worthwhile as we finally came up with a solution that we do not regard as too much of a compromise, but as a rather well-balanced and solid piece of design that might be of interest to others outside the consortium as well.

We did not attempt to invent any exciting or exotic new features, but instead aimed at a proper selection of important and well-established concepts and a good and seamless blend into a unified framework. Thus, the value of Chimera does not so much depend on the originality of its individual ingredients (though some new solutions have been developed, particularly wrt active rules), but on the way how these features have been combined and how potential conflicts have been resolved.

Chimera consists of a conceptual model (called Chimera Model, short: CM), providing object-oriented modelling facilities, and of a conceptual language (called Chimera Language, short: CL), providing data definitions, declarative queries, procedural primitives for database manipulation, as well as various forms of rules and constraints. Chimera has been designed in such a way, that it can either serve as an interface to a stand-alone database system or as a database sublanguage embedded in various procedural host languages. Due to the diversity of programming environments present in the IDEA consortium [ranging from C/C++ to PROLOG], aspects of procedural, general-purpose programming have been omitted from CL as far as possible. This issue has been left to the respective programming language into which Chimera will be embedded by the various partners. Certain basic linguistic decisions in CL, such as primitive value types, syntax of primitive values, conventions for distinguishing variables, constants or functors, and so on, may vary from one embedding to the other due to the particular conventions of the host language.

This paper is mainly aimed at providing a brief account of the spectrum covered by the Chimera design. The main ingredients of model and language are reviewed and some key decisions are highlighted. Representative examples, collected in the Appendix due to space limitations, will illustrate the flavor of the language.

2 CM: Chimera’s Data Model

First we will introduce Chimera’s data model in form of a “bird eye’s view” of the concepts to be found in CM. Each of the concepts is briefly introduced and the relationships between the individual parts of the model are sketched and discussed. A more precise, formal definition of the model (as well as of CL) is beyond the scope of this overview paper and can be found in [GBB94].
2.1 Objects and Values

CM is an object-oriented data model. Objects are abstractions of real world entities (such as persons, companies, or machines). Every entity of an application domain should be modeled as an object. Objects are distinguished from each other by means of unique object identifiers (OIDs), automatically generated by the DBMS on object creation. Objects are the essential components of a database, representing real world entities that are characteristic for the respective application domain. In contrast, values serve as a means of describing objects, but are not essential components of a database.

A value can be either atomic or structured. Atomic values are atomic, printable symbols, such as numbers or strings, or OIDs (i.e., references to objects). Structured values are built from atomic values by recursively applying one of the predefined constructors for sets, lists, or records. Chimera provides a number of predefined operators applicable to values, such as arithmetic operators (e.g., *, +, or sqrt) applicable to numbers, or selectors (like head and tail), applicable to constructs.

Attributes are functions mapping an object to a uniquely defined value. Any semantically meaningful information about a particular object has to be associated with that object by means of one of its attributes. Attribute values of an object may change over time, without changing the identity of that object. The collection of all attribute values associated with an object is called the state of that object. The state of an object is represented in Chimera as a record of attributes.

Objects can be manipulated by means of operations, defined and implemented by the designer of a particular application according to needs. An operation is essentially one of the procedural primitives offered by Chimera or a procedure written in the procedural host language of the environment in which Chimera is embedded. Rather than identifying Chimera operations and procedures one to one, a level of indirection has been introduced: operations are related to procedures by means of “guarded” declarations. When invoking an operation, the corresponding procedural code is executed only, if the “guard” (a declarative condition) is satisfied in the state of the database reached when the call is processed. In this way, operations are clearly separated from procedures conceptually, and control over operation invocations is assigned to the Chimera run-time system. Note that values cannot be described by means of attributes and cannot be manipulated by means of operations.

2.2 Types and Classes

Chimera supports both the notion of type and of class, with a one-to-one relationship between them. Whereas the notion of “type” emphasizes the structural and behavioural similarity of objects, the notion of “class” emphasizes membership of these objects in a common set of instances. In Chimera, most types remain implicit, while classes are defined, populated, and deleted explicitly, thus underlining the database character of Chimera: the main purpose of the language and the model is to provide an interface to a persistent store of collections of elements of the same type.

All objects of a Chimera application must belong to an object class. Classes must be defined first, and then objects may be inserted into classes.
Type definitions (i.e., definitions of structure and behaviour) are inferred automatically from class definitions. Whereas those attributes introduced in the previous subsection associate values with individual objects, Chimera provides a means for associating values with entire classes as well. **Class attributes** are functions mapping an entire class to a unique value. Examples of possible class attributes are cardinalities or statistical values such as average age or average salary for a class of person objects. Analogously, **class operations** manipulate an entire class rather than individual instances.

Object classes may be recursively specialized into **subclasses**, resulting in a taxonomic hierarchy of arbitrary depth. Multiple superclasses are possible, but multiple inheritance is subject to restrictions. A subclass inherits all attributes and operations from its superclasses, but may redefine their implementation. Moreover, a subclass may introduce additional attributes which are applicable to the objects in that subclass only. **Subclass definitions** are acceptable only if the corresponding (implicit) types are compatible. A set of subtyping conditions have to be satisfied whenever a class is defined as being subclass of another class.

Values are organized by means of types and classes as well. Most atomic values, such as integers, reals, or characters, are predefined and provide a (possibly infinite) “pool” of possible atomic symbols. These values cannot be manipulated by the database user, i.e., they cannot be inserted or deleted. Therefore, we do not regard concepts like `integer` or `string` as classes of Chimera, but as types only. Similarly, structured values are described by means of type constructors (record, set, and list) applied to value types. Structured value types also provide a (possibly infinite) “pool” of structured values. Once a type is defined, it is available in Chimera (for reuse in other object descriptions).

Chimera users may in addition introduce application specific collections of values, called **value classes**. These are populated explicitly, by means of insert operations, according to the needs of applications. In analogy to object classes, value classes are associated with a unique, but implicit type; they do not have attributes or operations. Value classes are used in Chimera as **active domains**: whenever the attribute of an object has a type which corresponds to a value class, the only allowed attribute values for that object must belong to the resp. value class.

### 2.3 Object Identifiers and Object References

In order to be able to express object-valued attributes as well as operations and constraints accepting objects as parameters, Chimera explicitly distinguishes between objects and object identifiers. **Object identifiers** are concrete symbols referencing a particular (abstract) object, but are not identical with that object. Thus, an object-valued attribute, e.g., managers of employees (being employees themselves) is in fact “OID-valued”. OIDs do not carry any meaning, but serve purely as internal surrogates. They are generated and manipulated by system software and cannot be retrieved by database users. In addition to OIDs, users of a Chimera application may, however, define their own **external names** for individual objects of particular interest.

This design choice leads naturally to an **object sharing semantics for object-valued attributes** (in which multiple objects may reference one and the same object by means of different attributes), rather than a copy semantics (in which
objects referred to by attributes of a given object would have to be copied in the state of that object).

### 2.4 Targeted and Untargeted Definitions

A schema definition in CL is a collection of targeted and untargeted definitions. Each type or class definition is a target, that is, a unit of abstraction and modularization. Features (such as attributes and operations) which are defined in the context of a given target have a scope that is limited to that target. Thus, targets enable a modular design and some degree of information hiding that is typical of object-oriented design (this is further supported by the separation between the definitions of signatures and implementations of each target, see later).

However, some information in the schema cannot be targeted; for instance, views combining information from several classes, or triggers affecting multiple classes, or constraints relating the state of objects from several classes. Therefore, some definitions cannot be expressed in the context of types and classes; these are called **untargeted definitions**.

Given that targeted definitions are usually easier to understand, control, and evolve, a good design principle for Chimera applications is to choose an appropriate collection of targets, so that most of the definitions in the schema can be targeted.

### 2.5 Derived Attributes, Derived Classes, Views

Passive rules of the CL language may be used for deriving information, as is normally done in deductive databases. Passive rules can be applied in the following situations:

- **Derived attributes**, can have values which are defined by means of rules instead of individual update operations over time. Derived attribute values are part of the object’s state as well.

- In the context of hierarchies of object classes, populations of some subclasses, called **derived classes**, may also be defined by means of passive rules, which generate the subclass population implicitly when certain properties of the superclass hold.

- Finally, untargeted **views** are always defined by means of passive rules which combine information from one or more classes of Chimera. Each view is introduced independently of a particular type or class definition and is “implemented” by one or more passive rules.

### 2.6 Constraints and Triggers

In addition to the basic structural and behavioural concepts of attribute and operation, Chimera provides two more concepts: constraint and trigger. Both concepts can be either targeted or untargeted.

**Constraints** are a means of restricting the contents of the database. Constraints consist of declarative conditions (expressed in CL). They have a name,
and they may have output parameters. In case of a violation of a constraint, these output parameters return values specific to the cause of the particular violation. Constraints that are targeted to a particular class may either restrict the extent of the respective class or restrict the set of legal values of its attributes; some constraints may be targeted to value types. Untargeted constraints restrict the set of legal database states and usually relate two or more classes. Constraints are checked after a commit command, issued by a transaction; if they are violated, then the transaction is rolled back.

**Triggers** are a means of introducing specific reactions to particular events relevant to the database. Such events are currently restricted to database specific operations (i.e. queries and updates) and operation calls. Other events, currently not supported in the language, could be time-related or external events. Reactions are calls to procedures written in either CL or the embedding host language. The execution of reactions is subject to conditions on the database state reached whenever an event is monitored; further, triggers are prioritized, so that when several of them can be fired a partial order is imposed. Triggers are named like constraints, but do not have parameters. A trigger that refers to a single class in each of its components (i.e., event, condition, and reaction part) can be introduced together with the definition of that particular class, as a targeted concept. All other triggers have to be introduced individually. Triggers are synonymously referred to as **active rules** within this document, because both notions are commonly used in the active database community, and in order to emphasize their relationship with passive rules (the former being imperative, the latter declarative specifications of certain autonomously initiated behaviour of the database system).

### 2.7 Signature and Implementation

When defining a new class, all attributes and operations of the new class have to be specified together with the corresponding domains (i.e., the class from which the attribute values or the operation parameters have to come). In addition, names and parameters of those constraints and triggers targeted to the particular class have to be introduced. The entirety of these name and domain definitions makes up the **signature** of the respective class.

Concepts are first introduced and “typed” in the signature of a class, then they have to be “implemented”. There are different ways of implementing a concept, depending on its particular nature. Classes can be populated either by explicit, individual creation of all instances, or by implicitly and collectively defining their instances by means of passive rules. The same applies in principle for attributes and class attributes: attribute values can either be introduced individually during object creation, or collectively by means of passive rules. Constraints (and class constraints) are always implemented by passive rules. Operations are implemented by means of a “guarded procedure body”, as mentioned earlier. Finally triggers are implemented by active rules. The association of attributes, classes, constraints, operations, and triggers with the expressions implementing them is performed after the respective signatures have been introduced. All concept definitions related to a particular class form the **implementation** of that class.

Both signature and implementation are considered part of the class definition. However, CL provides different data definition operations for introducing
signature and implementation, in order to reflect that implementations are usually introduced after signatures have been defined.

3 CL: Chimera’s Conceptual Language

CL serves two main purposes. It offers data definition facilities for implementing CM schemas (expressed by means of passive rules, active rules, and operation implementations). In addition, it provides for the manipulation of data by means of queries, updates and transactions (expressed by means of the declarative and procedural expressions of the language). CL comes in two slightly different versions, the one aimed at providing an interactive database language (supporting a “user-friendly interface” - UFI), the other to be used as embedded database sublanguage within a general-purpose programming language (forming an “application programming interface” - API). This section provides a brief summary of the concepts to be found in CL together with remarks concerning the rationale behind our design choices.

3.1 Declarative Expressions

CL is a logic-based language, supporting declarative queries, declarative (passive) rules for data definition as well as declarative conditions for controlling imperative (active) rules and imperative operations. Logical languages are classically composed of two main syntactic categories: terms and formulas.

**Terms** denote individuals in the respective domain of interpretation of the language. In the Chimera context this means that a CL term denotes either a value or an object. Atomic terms include constants and variables (starting with an uppercase letter). Complex terms are made from constructors (set, list, record) or by means of functions available in Chimera (attributes, selectors, predefined operators). Attributes are applied in postfix dot notation to variables denoting objects.

**Formulas** express propositions about individuals, being either true or false. In CL, there are five categories of atomic formulas:

- Class (and type) formulas are used for typing variables denoting objects or values.
- Membership formulas state that an object or a value is contained in a set.
- Comparison formulas provide means of comparing terms for equality or by means of a standard comparator.
- Constraint formulas serve as a means to access parameters of individual constraint violations (for compensation or repair, e.g.)
- Event formulas appear in conditions of active rules (see below)

Complex formulas are constructed from atomic ones by means of conjunction and negation. Formulas are evaluated over a database state according to the classical assumptions of first-order semantics. In order to avoid syntactically valid formulas which denote infinite set of instances, we impose that formulas be range-restricted. When compared to other Datalog-like logic languages, CL
offers a richer collection of mechanisms for building terms and formulas; these however enable us to implement and use all the features available through CM, which is a rich object-oriented data model.

3.2 Passive Rules

Passive rules are one of the key concepts of CL. They are used for declaratively defining class instances or attribute values, and for the implementation of views and constraints. A **passive rule** is an expression of the form $Head \leftarrow Body$ where the $Head$ is an atomic formula, the $Body$ is an arbitrary formula, and each variable in the head occurs positively in the body. Rules are stratified with respect to sets and negation, thereby ensuring that the computation of their fixpoint converges to a unique, minimal model. These limitations do not allow us to express certain semantics by means of rules (for instance, we exclude locally stratified rules). Moreover, our choice for a "standard" semantics for stratified rules excludes the possibility of choosing other semantics for more general rules, such as inflationary semantics. However, our design choice is motivated by the fact that stratified rules satisfy the requirements of most applications and have a more intuitive meaning than nonstratified rules.

3.3 Procedural Expressions

CL does not aim at being a full-fledged programming language. However, there is the need for expressing certain database-related imperative actions under the control of the database system, and thus for incorporating a certain degree of procedural syntax into CL.

Procedural expressions in CL are composed of primitive database statements, i.e., of updates, queries and operation calls. Chimera supports rather conventional notions of query, update, and transaction.

**Queries** in the UFI mode include *display* and *find*; queries supported from an API include *select* and *next*. In essence, all four query primitives are very similar. Each of them consists of a Chimera formula $F$ and a target list $T$. In all cases, the formula $F$ is evaluated over the current state of the database, returning either individual bindings to the variables in $T$ (in the case of *find* and *next*) or the set of all the bindings to these variables (in the case of *display* and *select*). Thus, the rationale of query primitives is to provide both instance-oriented and set-oriented access from both kinds of interfaces.

**Updates** in Chimera support object creation and deletion, object migration from one class to another, state change or change of persistency status of objects, and value class population and modification. This collection of primitives enables all possible class and persistency updates which can be envisioned in Chimera, with the only exception of turning an object from persistent to temporary.

The only general means of forming more complex statements is to build chains of primitive statements. Due to the database nature of these primitives, CL provides two different chaining operators: one for passing sets of variable bindings from one component statement to the other (the *sequence* operator) and one for passing individual bindings (the *pipe* operator). In addition, the UFI mode of interaction provides two syntactic variants which express iteration over sets of objects either explicitly or implicitly.
Finally, Chimera supports a conventional notion of **transaction**, where user-controlled **commit** and **rollback** primitives allow to either atomically execute all changes defined inside the transaction boundaries, or to restore the transaction’s initial state.

### 3.4 Active Rules and Operation Implementations

Similarly to the way how passive rules serve as a declarative means of implementing certain Chimera concepts, there are two categories of imperative constructs for implementing other concepts of CM: triggers are implemented by means of **active rules**, whereas operations are defined by means of an **operation implementation**. Both categories of constructs contain a procedural CL expression as their “body”, expressing a sequence of database actions that are to be executed. In both cases, trigger as well as operation, this execution only takes place, if a certain declarative (pre-)condition is satisfied over the current state of the database. The difference between the two categories of constructs is the style of invocation. In case of operations, an explicit invocation (operation call) is required and control is locally transferred from one operation call to the next. In case of triggers, invocation is implicit, controlled by a system component monitoring actions and determining appropriate reactions.

### 3.5 Triggers

Chimera supports **set-oriented triggers**, which are activated by database operations and perform reactive computations. The distinguishing choice of Chimera is to map an object-oriented data model with set-oriented triggers, e.g., triggers responding to collective operations. Triggers in Chimera follow the **event-condition-action** paradigm of active databases:

- **Events** correspond to database accesses for retrieval or manipulation
- Each **condition** is a declarative formula, to be evaluated in the state before activation of the reaction.
- Each **reaction** is a chain (sequence or pipeline) of one or more procedure calls, which can perform any computation on the database.

**Events** are a uniform interface for defining patterns of actions the observation of which can trigger a reaction. Such patterns can be:

- **Queries** performed over object classes. A query is any retrieval operation which occurs during the evaluation of a term in the context of a passive rule. Events are denoted by the name of the target of the query (either a class or an attribute of a class).
- **Updates** performed over object classes. Events are denoted by the name of the resp. update operation and the target (class name, possibly attribute name) of this operation.

Each trigger is defined on a set of triggering events; the set is associated to a disjunction semantics (the trigger becomes active if any of its triggering events occurs). We exclude, for the time being, to support within Chimera a
more complex event calculus. Note that we do not support triggers on value classes.

The condition is a declarative formula written in CL; it serves the purpose of monitoring the execution of the reaction part. The reaction is a chain of procedure calls; procedures can be either update primitives of Chimera, or display primitives, or operations, or externally defined procedures, or the transactional command rollback. Conditions and reactions may share some atomic variables that are used in order to relate them; in addition, conditions may use special formulas occurred and holds in order to identify objects which have been the target of one of the above events. Event formulas in the condition part of a trigger are provided as a means of referring to triggering events occurring within a transaction declaratively. Event formulas referring to updates are evaluated with respect to the net effect of a transaction. References to the state before and after a transaction can be made within the condition part as well.

Syntactically, active rules must be safe, that is, the variables occurring as input parameters of some procedures in the reaction part of the rule must be present in some positive literals of the condition part of the same rule (or be defined as output parameters of precedent procedures).

Execution of Chimera triggers may be controlled by means of a few control options (such as “coupling modes” or “event consumption modes”). Priorities can be given in order to partially order triggers relative to each other. A complete description of triggers in Chimera can be found in [CFPT94]; their formal semantics is presented in [FPT94].

4 Conclusion

In this paper we outline the main characteristics of Chimera, the conceptual interface to various implementations produced by the partners of the IDEA consortium within the ESPRIT III program of the CEC. Chimera consists of an object-oriented data model and a database sublanguage that supports a fully declarative query language, data definition and data manipulation primitives, passive and active rules, constraints and operations. Thus Chimera is one of the first attempts of combining concepts developed in active, deductive and object-oriented databases in a systematic manner. Chimera implementations are under development within the IDEA consortium and a rich variety of applications written in Chimera is currently designed and implemented by the application partners within IDEA. In addition, Chimera will be used as a standard conceptual model in the context of design methodologies and tools for supporting the development of active DOOD applications; assisting applications design will be a major objective of the second part of the IDEA project (June 1994 - June 1996).

Chimera is a joint effort of a group of researchers from various institutions. We would like to acknowledge the valuable contributions of Elena Baralis, Elisa Bertino, Stefano Crespi-Reghizzi, Claudia Roncancio, Christoph Draxler, Ulrike Greifahn, Piero Fraternali, Danilo Montesi, Stefano Paraboschi, Giuseppe Psaila, Andrea Sikeler, and Letizia Tanca.
Appendix

Due to space limitations we cannot include extensive examples in the text. Instead we provide a coherent set of example definitions in this appendix prototypically demonstrating all major concepts of CM and CL.

- An object class definition:

```
define object class person
attributes name:string(20)
birthday:date
vatCode:string(15);
age:integer
income:integer
profession:string(10)
operations changeIncome(Amount:integer)
constraints tooLowIncome(W:name)
key(V:vatCode)
c-attributes averageAge:integer
lifeExpectancy:integer
c-operations changeLifeExpectancy(Delta:integer):integer
c-constraints invalidLifeExpectancy(I:integer)
```

- Its implementation:

```
define implementation for person
attributes Self.age=X <= X=1994 - Self.birthday.year
operations changeIncome(Amount):
     integer(New), New=Self.income+Amount ->
     modify(person.income,Self,New)
constraints tooLowIncome(W) <=
    Self.income<500000, W=Self.name
C-attributes Class.averageAge=Y <=
    integer(Y), Y=avg(X.age where person(X))
C-operations changeLifeExpectancy(Delta):
    integer(New), Delta<10,
    New=Class.lifeExpectancy* Delta ->
    modify(person.lifeExpectancy,Class,New),
    return(New)
C-constraints invalidLifeExpectancy(I) <=
    I = Class.averageAge - Class.lifeExpectancy,
    abs(I) > 5
```

- A view on the class person:

```
define view marriage: record-of(husband:person, wife:person)
marrige((husband:X, wife:Y)) <= X.spouse=Y, X.sex=male
```
• An untargeted constraint:

```plaintext
define constraint tooLowLiability(Name: person.name, Plate: car.plate, Liability: integer)
  tooLowLiability(Name, Plate, Liability) <-
    person(X), car(Y), X = Y.owner, 
    Name = X.name, Plate = Y.plate, 
    Liability = Y.insuranceLiability*0.7, 
    Liability < 500000
```

• An untargeted trigger:

```plaintext
define trigger raiseBudget 
  events create(employee)
    modify(employee.salary)
    modify(dept.members)
    modify(dept.salaryBudget)
  condition dept(D), integer(I),
    I=sum(E.salary where employee(E), E in D.members),
    I>D.salaryBudget
  actions modify(dept.salaryBudget,D,I)
  after employee.adjustSalary
```

• A display query:

```plaintext
display(X.name,Y.make where person(X), car(Y), Y.owner=X)
```

• A create primitive:

```plaintext
create(project,
  (name="Intelligent DB Environment for
Advanced Applications",
  acronym="IDEA",
  number:6333,
  duration:4,
  commencement:(day:1,month:6,year:1992)),X
```

• A transaction:

```plaintext
begin transaction;
  select(X where employee(X), X.mgr.name="Manthey"),
  modify(employee.salary, X, X.salary + 5000);
commit
```

References