A Formal Definition of the Chimera Object-Oriented Data Model*

Giovanna Guerrini
Dipartimento di Informatica e
Scienze dell'Informazione
Università di Genova
Via Dodecaneso, 35 16146 Genova, Italy
`guerrini@disi.unige.it`

Elisa Bertino
Dipartimento di Scienze
dell'Informazione
Università di Milano
Via Comelico, 39 20133 Milano, Italy
`bertino@dsi.unimi.it`

René Bal
Computer Science Department
University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands
`rene@cs.utwente.nl`

Abstract

In this paper we formalize the object-oriented data model of the Chimera language. This language supports all the common features of object-oriented data models such as object identity, complex objects and user-defined operations, classes, inheritance. Constraints may be defined by means of deductive rules, used also to specify derived attributes. In addition, class attributes, operations, and constraints that collectively apply to classes are supported.

The main contribution of our work is to define a complete formal model for an object-oriented data model, and to address on a formal basis several issues deriving from the introduction of rules into an object-oriented data model.

1 Introduction

The area of data models has always been a very active research area that has a considerable impact on data management technologies. Indeed, the evolution of data models has marked the various generations of database management systems (DBMS). In particular, since the definition of the relational model, research has progressed to develop "new-generation" data models. A number of research and development directions in the data model area can be devised, among which the most relevant ones are object-oriented data models [13], deductive data models [21], extended relational models [48]. Research in those data models and related architectures have resulted in the development of several products and prototypes, such as the case of object-oriented DBMS (OODBMS) and extended relational DBMS. In general, although the various models are based on different concepts, there is now a large convergence towards models including the following features: complex objects; user-defined operations (e.g. methods, stored procedures); type hierarchies and

*The work of G. Guerrini and E. Bertino has been partially supported by the EEC under ESPRIT project 6333 IDEA. Project Esprit IDEA EP6333 participants are: Bull (prime), ECRC, ICRF, INRIA, Politecnico di Milano, TXT, University of Bonn.

inheritance; deductive and active rules, triggers. This trend clearly emerges if we look at recent work, like extensions to incorporate objects modeling capabilities into deductive databases [12], or to incorporate triggers and constraints into OODBMS [31], or to enhance SQL with object-orientation capabilities [30].

A system, whose goal is specifically the integration of object-oriented, deductive, and active capabilities, has been developed as part of the ESPRIT Project Idea P6333. The data model of Idea, called **Chimera**[1], is an object-oriented, deductive, active data model in that:

- it provides all concepts commonly ascribed to object-oriented data models, such as: object identity, complex objects and user-defined operations, classes, inheritance;
- it provides capabilities for defining deductive rules; deductive rules can be used to define views and integrity constraints, to formulate queries, to specify methods to compute derived information;
- it supports a powerful language for defining triggers.

In this paper, we present a formal definition of the Chimera data model. Such definition is quite a challenging task due to the many capabilities provided by Chimera. It is important to note that there has been a substantial theory work in the area of deductive data models. By contrast, no comparable amount of theoretical work has been reported in the area of object-oriented data models. Therefore, the main contribution of our work is to define a complete formal model for an object-oriented data model, and to address on a formal basis several issues deriving from the introduction of deductive and active rules into an object-oriented data model. A comparison of our formal model with other models is presented at the end of the paper.

This paper is focused on selected features of Chimera which specifically concern the type/class systems and their integration with operations. Its main contributions are the formal definition of the notion of schema and class, a precise characterization of type refinement, giving a set of typing rules for values, terms and formulas, together with a formal interpretation of the type system. However, this work does not cover the whole Chimera language; aspects of the language not modeled in this work are for example triggers and views. In the following sections we explicitly state the considered limitations, with respect to the generality of the Chimera language.

The remainder of this paper is organized as follows. The next section presents a brief overview of Chimera and a comparison of Chimera with other relevant systems. Section 3 deals with types and values, Section 4 with classes and objects. Section 5 deals with inheritance and subtyping, introducing also the notions of schema and database. The following step is the definition of Chimera rules, that are used to express class implementation. Section 6 is thus devoted to Chimera rules and their typing. Finally, Section 7 concludes the work, pointing out future work. The proofs of the results presented in this paper are omitted for brevity, they can be found in [35].

2 Overview of the Chimera Data Model

The main concepts of the Chimera data model are summarized as follows.

- **Objects:** are described by *attributes*, which can be *stored* or *derived* (e.g. defined by deductive rules), and are manipulated by *operations*, which can be *built-in* or *user-defined*. Each object

[1] A **Chimera** is monster of Greek mythology with a lion's head, a goat's body, and a serpent's tail; each of them represents one of the three components of the language.

is equipped with an object identifier (OID), which is unique within the entire the database, is generated by the system upon the object creation, and never changes.

- **Values:** both *primitive* and *complex* values are supported. Complex values are built by using constructors like set, list, record.

- **Classes and inheritance:** objects with similar structure and behavior are grouped in classes. Classes are organized in inheritance hierarchies. Multiple inheritance is supported. An object can belong to several classes, even classes not related by inheritance. Classes can be *explicitly populated* or *implicitly populated*. In the second case, a *population predicate* is associated with the class specifying sufficient and necessary conditions for an object to belong to the extent of the class. Upon creation of an object, instance of a class, the system may insert the object in some subclasses of this class if the object verifies the population predicates of these subclasses. Classes for which such a population predicate is not specified are said to be explicitly populated. Class-attributes and class-operations are supported, called in Chimera c-attributes and c-operations respectively. They are not inherited by objects of the class, rather they are used to characterize classes as objects.

- **Value types:** complex values are defined as instances of value types. Thus, value types provide the same function as concrete types commonly found in programming languages. A value type is *constrained*, if predicates are associated with the type definition specifying the legal values for the types, it is *unconstrained* otherwise. A value type is *extended* if its extent is explicitly defined (e.g. the user must specify the values belonging to the type), whereas the extent of other value types is implicitly defined. Therefore, extended value types provide the equivalent of enumerated domains.

- **Constraints and triggers:** constraints are predicates that must always be verified by the objects in the database. Triggers are used to specify actions to be executed whenever certain events and conditions arise. Both constraints and triggers are *targeted* if they are associated with a specific class, *untargeted* otherwise. Untargeted constraints and triggers usually involve several classes and therefore they are used to model inter-class constraints and triggers.

- **Views:** are similar to views of the relational model. Basically, a view is a query with which a name is associated and which is stored into the system catalogs.

We now elaborate on some of the above concepts and present examples to illustrate them. We refer the reader to [22] for additional details on Chimera.

Objects and Values

Chimera supports both values and objects. The main differences between those two notions in Chimera can be characterized as follows. First, objects are abstract, non-symbolic elements of the application domain; values are symbolic, printable elements. A second important difference is related to the notion of *identity*. Objects are described by attributes; however, their identity does not depend on the attribute values. Changing the values of an object's attributes does not change the object identity. A primitive value is identified by the value itself, whereas a complex value is identified by the values of all its components. Therefore, changing a component in a complex value changes the "identity" of the value. Finally, objects can be manipulated by user-defined operations, whereas values can only be manipulated via the pre-defined operations, which are provided by Chimera.

Chimera provides a number of constructors for building complex values, including the constructors set, list, and record. Constructors can be applied to: atomic values, complex values, object references. Therefore, constructors can be nested and a complex value may refer to an object.

Value types

A value type definition consists of a structure specification, which is mandatory, and a constraint specification, which is optional. A constraint specification, in turn, consists of the *constraint interface*, that is, the constraint name and the type of results, and of the *constraint implementation*. Constraints in Chimera are implemented as deductive rules. Those deductive rules have heads that may contain atoms with variables. Therefore, upon integrity checking a constraint may return values. Those return values are the values that have violated the constraint. Therefore, return values are used as the basis for explanation support in Chimera.

The following examples illustrate value type definitions expressed in the Chimera language. Example (3) illustrates the notion of extended value type.

1. ```
   define value type complex:
      record-of (re-part: integer,
                 im-part: integer)
   ```

   A value type, structured as a record of two components, is defined.

2. ```
   define value type dates:
      set-of(record-of (day: integer, month: integer, year: integer))
   ```

   A value type, structured as a set whose elements are records consisting of three components, is defined. This definition exemplifies nesting of constructors.

3. (a)
   ```
   define value type postalCode: integer
      constraints improperCode (Code: postalCode)

   define implementation for postalCode
      constraints
         improperCode(Code) ← Code < 0;
         improperCode(Code) ← Code > 9999
   ```

   (b)
   ```
   define extended type postalCode: integer
      constraints improperCode (Code: postalCode)

   define implementation for postalCode
      constraints
         improperCode(Code) ← Code < 0;
         improperCode(Code) ← Code > 9999
   ```

   In the above example, both definitions of ‘postalCode’ have the same implementation for the constraint with name ‘improperCode’. Note that when definition (3.a) is used, the valid values are all 4-digit integers in the range (0, 9999). By contrast, when definition (3.b) is used, the valid values are all 4-digit integers in the range (0, 9999) explicitly inserted into the type extent. Therefore, only the values that are inserted[2] by the users into the extended type `postal code` are valid values for this type.

[2] Chimera provides an insert operation for adding values into extended types. Such operation is not defined for other value types.

Therefore, the use of value types, extended value types and constrained value type in Chimera supports: (*i*) domain definition through range specification; (*ii*) domain definition through explicit enumeration; (*iii*) definition of validity conditions for values. A large number of modeling capabilities for domains can thus be supported.

Classes

A class definition in Chimera consists of two components: the *signature*, specifying all information for the use of the instances of the class; the *implementation*, providing an implementation for the signature. Furthermore, each class is associated with an *extent* collecting all instances of the class. The signature consists of a number of clauses, including the specification of: attributes, operations, constraints, class-attributes, class-operations, class-constraints, triggers and superclasses. The signature also specifies whether the class is implicitly populated or not, and for each attribute whether the attribute is derived or not. The implementation of a class must specify an implementation for all derived attributes, derived class-attributes, operations, class-operations, constraints, class-constraints, and triggers that are specified in the signature. Note that the implementation of operations may be specified by an update rule, that is a rule with update atoms, or may be external, implemented in some programming language[3]. The following example shows a class signature and implementation.

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

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(N) ← Self.income < 5000, N = 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)
   c-constraints  invalidLifeExpectancy(I) ←
                  I = Class.averageAge - Class.lifeExpectancy, abs(I) > 5
```

Classes are organized into inheritance hierarchies. The subclass compatibility conditions will be discussed in the following subsection. The current version of the language does not allow redefinition of constraints and triggers in subclasses.

[3] Idea provides the Peplom language for this purpose [27].

2.1 Typing in Chimera

In Chimera, a subclass may redefine an inherited feature but the redefinition is subjected to the following constraints:

- an attribute may be redefined in a subclass by specializing (or refining) its domain, that is, replacing its domain by a proper subtype (*covariant* redefinition);
- an operation may be redefined in a subclass by specializing its signature; signature specialization may be obtained by replacing at least one of the domains of the input parameters with a proper supertype or at least one of the domains of the output parameters with a proper subtype (*contravariant* redefinition for input parameters, *covariant* redefinition for output ones).

These choices clearly have an impact on Chimera policies with respect to type checking. While the redefinition of operations does not hinder the type safety of the language, the redefinition of attributes must be considered carefully. This covariant redefinition of attributes (the domain of an attribute may be specialized in subclasses) reflects what is usually needed when creating a taxonomy of classes; indeed, when specializing a class the designer usually needs to to add new attributes or to specialize existing ones. The problems arising when attributes are redefined in a covariant way along the inheritance hierarchy has been firstly recognized by Cardelli [17] and are shown by the following example.

Example 1 *Consider a class* `person` *with an attribute* `spouse` *with domain* `person` *and a class* `noble` *(subclass of* `person`*) which refines the domain of the attribute* `spouse` *in* `noble`[4]*. Suppose that in class* `person` *an operation* `set-spouse` *is defined, which takes as input a value of type* `person` *and assigns that value to attribute* `spouse`*. This operation is not redefined in class* `noble`*. Now consider two variables* `P`, `N` *both declared of type* `person`*. The expression* `N.set-spouse(P)` *is statically type correct but if run-time* `N` *is instantiated to a noble and* `P` *to a (not noble) person, it results in trying to assign a value of type* `person` *to an attribute whose domain is* `noble`*.* ◇

The approach adopted in Chimera is to consider the domains of attributes as integrity constraints, thus checked run-time, rather than dealing with them as type constraints, to be checked statically. Thus, whenever a value is assigned to an object attribute we dynamically check that the value is appropriate for the domain.

Of course, there are other possible alternatives for dealing with covariantly redefined attributes. The approach taken by Cardelli in [17] and adopted by Galileo [3] is to syntactically distinguish among mutable (assignable) and immutable (not assignable) entities and to disallow covariant redefinition for mutable entities (immutable entities may receive their value at object creation only). Thus, all the attributes that must be redefined cannot be updated; on the contrary, if an attribute should change its value during the lifetime of its enclosing object, then it may only be inherited, but not redefined, by subclasses. This solution is obviously not the most adequate for the database context, in which objects are long-living evolving entities.

Another possible solution to the problem is to limit the context in which subtype substitution is allowed. A proposal along this way was made by Connor and Morrison [26]; their approach guarantees static type safety at the expense of limiting the expressiveness of the language and introducing a lot more of typing information inside the program. Note moreover that in their type system, type specifications are exact.

[4]Here, more or less realistically, we suppose that the spouse of a noble is always a noble.

The Eiffel language [43] allows the covariant redefinition of attributes and catches every attempt of incorrect assignment at program link time, using an approximate and pessimistic algorithm which computes an approximation of the possible types of the objects that may be referenced by each entity at each point of the program where the entity is used. The TDF technique [25] implements a similar concept, but differs in the algorithm used to compute the type sets: TDF adopts a data-flow technique, while Eiffel's global verification does not take into account the possible flow of control of the program. Both techniques, though, loose much of their precision when persistent data are involved, since it is not possible to determine the types of the objects referenced by persistent variables. In the example, the bindings for variables `N` and `P` may be taken from the extents of classes manipulated by programs different from the one examined.

The approach of O_2 is the closer to ours. O_2, indeed, does not restrict either the assignment or the covariant redefinition of attributes and checks the validity of every assignment dynamically, possibly generating a run-time error. All these approaches to the covariant redefinition of attributes are extensively reviewed in [19].

Another feature of the Chimera language that may cause some problem with respect to type checking is related to derived classes, as shown by the following example.

Example 2 *Consider a class* `employee` *with a derived subclass* `rich-employee` *with the population constraint* `X.salary > 5000`. *Consider, moreover, a variable* `X` *defined with type* `rich-employee`. *If the value of* `X.salary` *is modified to* `4000`, `X` *can no longer be an instance of the class* `rich-employee`. *This leads to an inconsistency.* ◇

To avoid such problem, different solutions can be adopted. Two of them, namely

- disallowing a variable to be declared with a type corresponding to a derived class;
- disallowing updates on the attributes appearing in the population predicate

are conservative solutions. If the population predicate can only be falsified by updates on the object on which it is evaluated[5] the above solutions can be refined by allowing a variable to be declared with a type corresponding to a derived class but disallowing updates on the attributes appearing in the population predicate to be applied to that variable. Such an approach prevents, by static checks, a variable of type T from referencing at run-time an object which is not an instance of the class corresponding to T. This approach emphasizes the type checking view.

The solution currently adopted in Chimera is to regard population constraints as other constraints and thus to check them run-time. This approach does not ensure that a variable of type T, with T corresponding to a derived class, always references a member of the class corresponding to T. Rather a check is performed at run-time to detect whether the variable references an object that meets the population constraint. If not, an error is raised.

This approach requires some type checking at run-time and thus it is potentially less efficient. Note however, that the two above solutions are not mutually exclusive. They can be combined to obtain a good compromise between semantic richness and efficiency. For example, a variable can be allowed to be declared of a type corresponding to a derived class, and updates on the attributes appearing in the population predicate can also be applied to that variable, but run-time checks for that variable (and only for that one) must be performed. More sophisticated solutions, based on flow analysis of application code [25], can also be investigated.

[5]This is not always true, e.g. if the population predicate makes use of aggregate operators.

To conclude this section, let us mention that the "contravariant for input, covariant for output" rule for the redefinition of operation signatures ensures that the new signature may be used everywhere the old one can be used, and guarantees that no type errors occur due to the redefinition of operations. There has been a long debate on whether this "contravariant for input" rule, besides being type safe, is also intuitive to use. The possibility of allowing a covariant redefinition of input parameters of operations is currently under investigation in Chimera [19].

2.2 Comparison of Chimera with other Object-Oriented Data Models

Table 1 summarizes some of the differences of Chimera with the data models of other OODBMS and of the ODMG object database standard. In particular, a difference among the various systems is whether the class extents are managed by the system or must be managed by the application. In the latter case, applications must define "object collections", by using a constructor like the set constructor, to group the instances of the same class and must explicitly manage the addition/removal of objects from these collections. Multiple collections over the same class can be defined. The notion of composite object refers to the semantic notion of "part-of" [39]. This notion, only supported by the Orion system, models the fact that an object (called *component object*) is part of another object (called *composite object*). This semantic notion may influence the way certain operations are performed; for example, the deletion of a composite object may imply the deletion of all its component objects. We refer the reader to [39] for additional details on composite object models. A composite object model for Chimera is being developed [34].

In general integrity constraints are not supported. One reason is that they can be implemented into the method codes. In Chimera a more declarative approach is used by which a declarative constraint language is provided. Apart from Chimera, the Iris system provides a limited form of constraint through the use of the key/nonkey qualifiers. Those qualifiers are associated with attribute definitions. For example, if an attribute is defined to be a key, all instances of the same class must have different values for this attribute. Referential integrity is currently not directly supported in Chimera. However, it can be enforced through the use of triggers. To rely the user from the burden of explicitly defining triggers for enforcing referential integrity in [8] we have shown how triggers can be automatically derived from declarative specifications of actions to be taken upon violation of referential integrity (the considered actions are those supported by SQL2 [15], i.e. `CASCADE`, `SETNULL`, `RESTRICT`).

We refer the reader to [13] for an extensive discussion about the several variations in object-oriented data models.

3 Types and Values

In Chimera both the notion of type and the notion of class are provided. The notion of type in Chimera is similar to that of concrete (structural) type. The notion of class is similar to that of an abstract data type coupled with an extent[6]. Note, however, that full encapsulation of object structures is not enforced in Chimera.

In Chimera each class is related to a type (which we call type of the class) which describes the structure of the objects belonging to the class. In order to type variables that have to be instantiated with objects belonging to a given class, we allow the use of a class name as a type. A value type is a type in the previously introduced meaning, whereas an object type is the type

[6]We refer to the notion of concrete and abstract types as used in Galileo [3].

	Chimera	GemStone	Iris	O2	Orion	TM	Ode	Cocoon	ODMG
Reference	[22, 23]	[14]	[29]	[28]	[40]	[5]	[2]	[47]	[20]
User-defined value types	YES	NO	NO	YES	NO	YES	NO	NO	YES
Extended value types	YES	NO	NO	NO	NO	NO(1)	NO	NO	NO
Class extent	System	User	System	User	System	System	User	System	System
Class features	YES	YES	NO	NO	YES	YES	NO	NO	NO
Composite objects	NO(2)	NO	NO	NO	YES	NO	NO	NO	NO
Referential integrity	NO(2)	YES	NO	YES	NO	YES	YES	YES	YES(3)
Multiple inheritance	YES	NO	YES	YES	YES	YES	YES	YES	YES
Intensionally defined attributes	YES	NO	YES	NO	NO	NO	NO	NO	NO
Declarative method implementation	YES	NO	YES	NO	NO	YES	NO	NO	NO
Integrity constraints	YES	NO	L.F.	NO	NO	YES	YES	L.F.	NO
Triggers	YES	NO	NO	NO	NO	NO	YES	NO	NO
Views	YES	NO	YES(4)	NO	NO	NO(1)	NO	YES	NO

Legenda: L.F. limited form
(1) Extended value types and views could be simulated in TM by using the module mechanism provided by this language [5].
(2) Actually, Chimera is being extended with the support for composite objects and referential integrity [34].
(3) Referential integrity is enforced for relationships, but it is not for attributes.
(4) Views in Iris are supported as functions implemented in OSQL which are not associated with a specific type.

Table 1: **Comparison with other OO data models**

corresponding to a class (the class name). We provide a uniform notion of type (regarding class names as types) to handle in uniform way type checking.

3.1 Types

We postulate the existence of a collection of basic domains $D_1, \ldots, D_n$ to which at least the types `integer`, `real`, `bool`, `character` and `string` belong. Furthermore, let $\mathcal{OI}$ denote a set of object identifiers, and $\mathcal{CI}$ a set of class identifiers. The elements of $\mathcal{OI}$ are identifiers of objects, used as a mean to refer to and distinguish objects in the system. By contrast, $\mathcal{CI}$ contains class identifiers, that is, class names. In the following we make use of a set of type names $\mathcal{TN}$, a set of attribute names $\mathcal{AN}$ and a set of method names $\mathcal{MN}$. We assume all the introduced sets to be mutually disjoint[7]. For easy of reference, Table 2 illustrates the symbols more frequently used in this paper. For each symbol, the table reports a brief explanation of its meaning.

In the following, we let i vary over $\mathcal{OI}$, c over $\mathcal{CI}$, Tn over $\mathcal{TN}$ and a over $\mathcal{AN}$. The following

[7] This assumption is only useful to simplify definitions and results, but it is used only at a theoretical level, while in the language it does not hold. One of the most remarkable characteristics of object-oriented data models is indeed the independence of each class definition from the others.

Symbol	Meaning
$D_1, \ldots, D_n$	basic domains
$\mathcal{OI}$, $\mathcal{CI}$	a set of object/class identifiers
$\mathcal{TN}$, $\mathcal{AN}$, $\mathcal{MN}$	a set of type/attribute/method names
$\mathcal{T}$, $\mathcal{OT}$, $\mathcal{VT}$,	the set of Chimera types, object types, value types
$\mathcal{BVT}$, $\mathcal{CT}$, $\mathcal{ET}$	the set of basic value types, constrained types, extended types
$\mathcal{V}$	the set of Chimera legal values
Var, Var_T	the set of variables and variables of type T
$type(T)$	the type obtained from T replacing each named type appearing in it with the type it stands for
$struct(T)$	the structural component of the constrained or extended type T
$constr(T)$	the constraint component of the constrained type T
$ext(T)$	the extent of the extended type T
$T_1 \sqcup T_2$	the most specific common supertype of T_1 and T_2
$\pi(c)$ $(\pi_O(c))$	the extent of class c
$\pi(T)$ $(\pi_V(T))$	the extent of the extended type T
$dom(D_i)$	the (postulated) non-empty extension of the basic value type D_i
$stype(c)$	the type of the state of objects of class c (the *structural type* of class c)
$value(i)$	the state of the object identified by the oid i
$v : T$	the value v belongs to type T
$[\![T]\!]_\pi$	the extension of type T under the assignment π
$o_1 = o_2$	objects o_1 and o_2 are equal by identity
$o_1 == o_2$, $o_1 ==_d o_2$	objects o_1 and o_2 are equal by shallow/deep value equality
$ISA(c)$, $ISA^*(c)$	the set of the direct/of all the superclasses of class c
$ISA_V(T)$, $ISA^*_V(T)$	the set of the direct/of all the supertypes of the value type T
$T_2 \leq_T T_1$	T_2 is a subtype of T_1
$s \leq_S s'$	signature s is a refinement of signature s'
$MC_2 \leq_{MC} MC_1$	metaclass MC_2 is a refinement of metaclass MC_1
$C_2 \leq^\iota_C C_1$, $C_2 \leq^e_C C_1$	class C_2 intensionally/extensionally refines class C_1

Table 2: **Notations and terminology**

definitions formalize the various notions of types of the Chimera language.

Definition 1 *(Predefined Basic Value Types). The set of predefined basic value types $\mathcal{BVT}$ is $\{D_1, \ldots, D_n\}$, that is, it is the collection of all the basic domains.* □

In Chimera, class names can be used in definitions of types. This is due to the fact that attributes of types, structured as records, are allowed to have classes as domains. The definition of collections, structured as sets or lists, of instances of classes must be supported in Chimera. Those collections are used, for example, to store query results.

The following definition states that each class name is a (object) type, thus introducing a uniform notion of type.

Definition 2 *(Object Types). The set of Chimera object types $\mathcal{OT}$ is defined as the set of class identifiers $\mathcal{CI}$.* □

The distinction between value and object types is needed in order to distinguish types, whose instances are object identifiers in $\mathcal{OI}$ (namely values used to identify objects), from types, whose instances are pure values. The distinction between objects and values is a rather subtle issue of object-oriented data models. According to Beeri [7], the difference between these two notions can be characterized as follows: values are used to describe other entities, whereas objects are the entities being described. That is, the information carried by a value is the value itself, whereas the relevant information about objects is carried in the relationships they have with other objects and values. As a consequence, objects have an identifier and a state, which is a value, representing the information carried by the object. By contrast, the identifier of a value is the value itself. Therefore we may say that value types are types whose instances do not have an explicit identifier, whereas object types are types whose instances have explicit identifiers.

In this respect Chimera is rather similar to the O_2 data model [28, 41] which supports both objects and complex values, where objects are (*identifier, value*) pairs and values are built using atomic values, structured values and object identifiers. The following definition introduces Chimera value types.

Definition 3 *(Value Types). The set of Chimera value types $\mathcal{VT}$ is inductively defined as follows*

- *the predefined basic value types are value types ($\mathcal{BVT} \subseteq \mathcal{VT}$);*
- *if T is a value type or an object type then*

  $list\text{-}of(T)$ *and*
  $set\text{-}of(T)$

  are value types called (predefined) structured value types, respectively indicated as list type and set type;
- *if $T_1, \ldots, T_n$ are value types or object types and $a_1, \ldots, a_n$ are distinct labels in $\mathcal{AN}$[8], then*

  $record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)$

  is a value type called (predefined) structured value type, indicated as record type;

[8] Note that the possible labels in record types are attribute names.

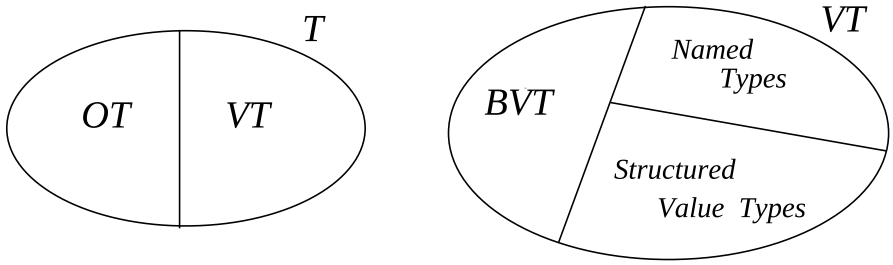

Figure 1: Chimera types

- *if T is a value type (either basic or structured) and Tn is a type name in $\mathcal{TN}$, and Tn is declared as a name for T through a declaration*

  $Tn : T$

  then Tn is a value type called user-defined (named) value type. □

Example 3 *The following are Chimera value types*

```
integer
boolean
set-of(character)
list-of(boolean)
record-of(re:real, im:real)
set-of(record-of(day:integer, month:integer, year:integer))
```

`complex` *and* `dates`*, introduced by the following type declarations, are (named) value types:*

```
complex:  record-of(re:real, im:real)
dates:  set-of(record-of(day:integer, month:integer, year:integer)).
```

Finally, if `person` *belongs to $\mathcal{CI}$, then*

```
set-of(person)
record-of(n:string, s:person)
```

are (structured) value types. ◇

According to the above definition, the set of predefined value types is an infinite set. Note, however, that each schema contains only a finite subset of predefined value types, that is, those types that are named or used somewhere in the schema. Therefore a Chimera schema is always finite.

The set of Chimera types $\mathcal{T}$ is thus defined as the union of Chimera value types $\mathcal{VT}$ and Chimera object types $\mathcal{OT}$. A Chimera type is therefore either a Chimera value type or a Chimera object type. We provide the notion of type because in some situations we want to talk of "type" disregarding whether it is a value type or an object type. Figure 1 depicts Chimera types, emphasizing that the set of object types and the set of value types partition the set of Chimera types, that is, they are disjoint and their union is the set of Chimera types $\mathcal{T}$.

Example 4 *Consider the class names* `person` *and* `department`*, both belonging to $\mathcal{CI}$. Then the types* `person` *and* `department` *are examples of object types, whereas the types*

```
record-of(name:string, dep:department)
list-of(person)
```

are value types. ◇

Type names are assigned only to value types, not to object types. The same name cannot be assigned to two different types. That is, a name introduced by a declaration of the form $Tn : T$ (according to the last item in Definition 3) cannot be redeclared as $Tn : T'$. This condition ensures that the function *type*, defined below, is really a function, that is, the value type associated with a name is unique. Note that named types, basic value types and structured value types are disjoint sets, and they partition the set $\mathcal{VT}$ of Chimera value types. Each type in $\mathcal{VT}$, indeed, is either a named type, or a basic value type, or a structured type. As we will discuss in Section 5 inheritance relationships link named types and basic value types or named types and structured value types. By contrast, object types and value types constitute disjoint inheritance hierarchies.

Named value types are introduced in Chimera as a mean to bind a type name to a value type. This mechanism is useful for future references to the, possibly structured, type and for type renaming. Note, however, that a name is assigned to a type only to simplify references to that type and that we have complete equivalence and substitutability of the type with respect to the name. Therefore, we disregard type names when considering type equivalence and compatibility. For this purpose, a (total) function $type : \mathcal{T} \rightarrow \mathcal{T}$ is defined, that for each type T, returns the type obtained from T by substituting each type name with the value type identified by the name. This function only affects value types. However, it has been defined on all types because object types may be components of value types.

Definition 4 *Let $T \in \mathcal{T}$ be a Chimera type. Then*

$$type(T) = \begin{cases} T & \text{if } T \in \mathcal{BVT} \text{ or } T \in \mathcal{OT} \\ list\text{-}of(type(T')) & \text{if } T = list\text{-}of(T') \\ set\text{-}of(type(T')) & \text{if } T = set\text{-}of(T') \\ record\text{-}of(a_1 : type(T_1), \ldots, a_n : type(T_n)) & \text{if } T = record\text{-}of(a_1 : T_1, \ldots, a_n : T_n) \\ type(T') & \text{if } T \in \mathcal{TN} \text{ and it has been defined} \\ & \text{by a declaration } T : T' \end{cases}$$

□

The function has been defined by induction on the type structure of T by recursively substituting each type name in a value type with the value type identified by the name (also as component of structured types) and to deal also with named types defined in terms of other named types.

Example 5 *Consider the following Chimera type definitions.*

```
nat:  integer
pos:  integer
np-rec:  record-of(a:nat, b:pos)
s:  set-of(np-rec)
```

Then

type(`nat`) = *type*(`pos`) = `integer`
type(`np-rec`) = `record-of(a:integer, b:integer)`
type(`s`) = `set-of(record-of(a:integer, b:integer))` *type*(`list-of(np-rec)`) = `list-of(record-of(a:integer, b:integer))`. ◇

3.1.1 Constrained Types

In addition to user-defined value types introduced as synonyms for (and abbreviations of) predefined types, Chimera provides the possibility of defining types denoting a proper subset of the set of values of a user-defined value type. In such cases, types are restricted by means of constraints expressing conditions on the values of the types. Types, defined by restriction on other types, are called *constrained types*. A constrained user-defined value type is a named type for which not only a structure but also a constraint is specified. Constraints associated with constrained types are expressed as Chimera rules, whose definition is presented in Section 6. Intuitively, a Chimera rule is a deductive rule containing in its head an atom built using predicate symbols and whose body consists of a conjunction of atoms or negated atoms. Atoms are built using predicate symbols. A number of predefined predicate symbols exist in Chimera, including the equality and set/list membership. Rules are implicitly universally quantified as in Datalog. Rules used to express constraints for a constrained type have a constraint atom in their head, where a constraint atom is an atom built applying a constraint name to a single argument whose type is the value type itself.

Definition 5 *(Constrained User-defined Types). If $Tn \in \mathcal{TN}$ is a type name, $T \in \mathcal{VT}$ is a value type, R is a set of Chimera rules and Tn is declared as a name for the type whose structure is T and whose constraint is R, by a declaration*

$Tn : T$ `constraint` R

then Tn is a constrained user-defined type. In the following struct(Tn) denotes the value type T (which is the structural component of the type, describing the structure of type elements) and constr(Tn) denotes the set of rules R. □

Rules in R express a condition that must not be verified by the values of the type. Constraints in Chimera are indeed expressed in denial form. Therefore, they state what should not be the case for any legal database state, rather than expressing invariant conditions of the database. In case of constraint violation, the parameter(s) of the constraint are bound to values characterizing the "violated" instance of the constraint.

Example 6 *Consider the type name* `postalCode`*, the basic value type* `integer` *and the following set R of rules:*

{ `improperCode(Code)` ← `Code` < 0
`improperCode(Code)` ← `Code` > 9999 }

Then the type declaration `postalCode`*:* `integer constraint` *R defines* `postalCode` *to be a constrained value type.* ◇

Let $\mathcal{CT}$ denote the set of constrained user-defined types. Constrained user-defined value type are a special case of named types. Indeed, with a type name not only a value type is associated but also a constraint which expresses restrictions on the value type. Thus

$$\mathcal{CT} \subseteq \mathcal{TN} \subseteq \mathcal{VT} \subseteq \mathcal{T}.$$

Note that, unlike named value types, constrained types are not equivalent to the type structure assigned to the name, rather they are a proper subtype of that type. Indeed the set of values of a constrained type is a subset of the set of values of the type structure assigned to that type.

Thus, when talking of constrained types the complete substitutability of the type with respect to the name is lost. By contrast, a constrained type is a subtype of the value type that describes its structure, in that all the legal values of the former are also legal values for the latter. Consider for example the value type

```
record-of(day:integer, month:integer, year:integer).
```

Now consider the constrained type `date`, such that

$struct$(`date`) = `record-of(day:integer, month:integer, year:integer)`
$constr$(`date`) = { `improperDate(Date)` ← `day ≤ 0`
`improperDate(Date)` ← `day > 31`
`improperDate(Date)` ← `month ≤ 0`
`improperDate(Date)` ← `month > 12`
`improperDate(Date)` ← `year ≤ 0` }

Obviously, each date is a legal value for the value type `record-of(day:integer, month:integer, year:integer)`, but the converse is not true. Thus `date` is a subtype of that type.

Similarly, if we consider the constrained type `summerdate`, specified as

$struct$(`summerdate`) = `date`
$constr$(`summerdate`) = { `improperSDate(SDate)` ← `month < 6`
`improperSDate(SDate)` ← `month > 8` }

then `summerdate` is a subtype of `date`.

Therefore, a partial ordering on value types exists such that unconstrained value types are not comparable under the ordering. By contrast, each constrained type is related to an unconstrained type (of which it is smaller) which is the type that describes its structure (the type returned by the *type* function[9]). We formalize these notions in Section 5 when defining an ordering on types.

3.2 Extended Types

In addition to predefined and user-defined value types, Chimera provides the notion of extended type. The notion of extended value type can be characterized as follows. Predefined and user-defined value types are considered as "abstract domains" in Chimera, in the sense that the set of instances of the type is never made explicit. An explicit enumeration of the individual instances of the type is not possible. By contrast, extended value types are used to control the extent of a user-defined value type by enumerating its values. When defining a extended value type not only a value type with the same name and definition is generated, but also the storage for an explicit extent is allocated. Thus extended value types are a notion rather similar to that of enumeration types of programming languages (e.g. Pascal) but with two differences: (*i*) in a extended value type not only the explicit enumeration is provided but also a value type that specifies the structure of values of the (explicit) extent; (*ii*) the extent of a extended value type is not fixed, rather it can be modified. Constraints may be defined on extended value types as well, thus restricting their explicit extents.

Definition 6 *(Extended Value Type). Let $ET \in \mathcal{TN}$ be a type name, $T \in \mathcal{VT}$ be a value type, S be a set of values of type T and T and S be associated with ET by a declaration*

[9] The *type* function of Definition 4 is easily extended to constrained types by defining $type(T) = type(struct(T))$ whenever $T \in \mathcal{CT}$.

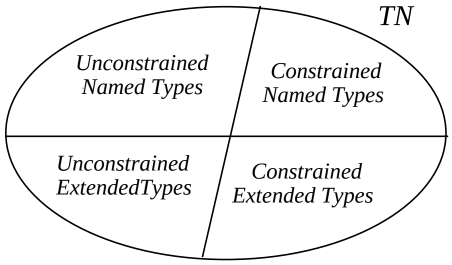

Figure 2: Chimera named types

$ET : T$ `ext` S

then ET is a extended value type. If type T is a constrained type, then ET is said constrained, it is said unconstrained otherwise. In the following $struct(ET)$ denotes the value type T (which is the type describing the structure of class elements) and $ext(ET)$ denotes the set S which maintains the explicit, time-varying extent of class ET. $\Box$

Example 7 *Consider the type name* `colour` $\in \mathcal{TN}$*, the basic value type* `string` *and the set* $S =$ {`green, blue, red, yellow`}*. Then the type declaration* `colour`*:* `string ext` S *defines* `colour` *as a (unconstrained) extended value type.*

Consider now the type name `realpostalcode` $\in \mathcal{TN}$*, the value type* `postalcode` *of Example 6 and the set* $S =$ {`5574, 7982, 6827, 1343`}*. Then the type declaration* `realpostalcode` *:* `postalcode ext` S *defines* `realpostalcode` *as a (constrained) extended value type.* $\diamond$

Let $\mathcal{ET}$ denote the set of extended value types. The inclusion relationships

$$\mathcal{ET} \subseteq \mathcal{TN} \subseteq \mathcal{VT} \subseteq \mathcal{T}$$

hold. Extended value types are types rather than classes, though they have an explicitly enumerated collection of instances, that is, an explicit extent, since their instances are values, and not objects. They may indeed be seen also as a sort of enumeration types.

The partitioning of Chimera named value types $\mathcal{TN}$ is depicted in Figure 2. Note that the sets of constrained extended value types, unconstrained extended value types, constrained value types and unconstrained value types are disjoint sets, but, however, inheritance relationships can exist between types in different sets, as we will discuss in Section 5.

Note that the extent S of a extended value type is not a fixed set, rather it can be updated by means of `add` and `drop` operations. Therefore, a extended value type could be regarded as a class (cf. Section 4.1) in that it consists of a type, an explicit extent and two predefined update operations. This is the reason why extended types are called *value classes* in Chimera [22]. However, user-defined methods, triggers and other features of object classes are not supported for extended value types. This is the reason why we have preferred to refer to them as types.

The `drop` operation on extended value types, though well handled at a theoretical level, poses implementation problems. Indeed it must be regarded as a schema update. Consider a class C having as domain for an attribute a extended value type ET with which an extent S is associated. Upon each object creation in class C the correctness of the attribute value is checked with respect to the extended value type ET. A value v is correct for the attribute, having as domain ET, if it is a member of S. If value v is then dropped from S, the attribute value is no more legal. As a consequence, after any drop operation on the the extent of ET, the legality of attribute values must be checked or we may have an inconsistent database.

Note that we have ISA relationships when considering extended value types, as well. Indeed, if a extended value type ET is defined in terms of a value type T, then all the elements of ET are elements of T, while the converse is not true (unless $ext(ET)$ coincides with the whole extension of type T). Thus, the extended value type is a subtype of the value type which specifies its structure, that is $struct(ET)$. Referring to the example above, `colour` is a subtype of the type `string`, while `realpostalcode` is a subtype of the type `postalcode`.

We remark that we cannot define extended value types in terms of extended value types, that is, $\forall\ ET \in \mathcal{ET},\ struct(ET) \in \mathcal{VT} \setminus \mathcal{ET}$, thus ISA relationships among extended value types do not hold.

3.3 Most Specific Common Supertypes

In this section we introduce the notion of *most specific common supertype (mscst)*, which will be used when defining legal values for types. This notion is introduced because, when regarding class names as types and when considering constrained types and extended value types, type extensions are not disjoint due to ISA relationships among classes and among types. Therefore, two different types may have a common supertype. In such a case, values, instances of those types, can be regarded as values of the *mscst*, and therefore these values are compatible.

First of all, we address the issue of type equality. Two types are equal either if they are identical or if they are record types obtained by permuting their fields. Indeed, the order of fields in record types is immaterial. Therefore the value type

$$record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)$$

is equal to the type

$$record\text{-}of(a_{j(1)} : T_{j(1)}, \ldots, a_{j(n)} : T_{j(n)})$$

where $j : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ is a bijective function. Two named types are equal only if they have the same name.

Intuitively, the most specific common supertype of a set of types is the smallest type being a supertype of all types in the set. The operator $\sqcup$ is a partial function, $\sqcup : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$ returning the most specific common supertype between two types.

Definition 7 *(Most Specific Common Supertype). Let T_1 and T_2 be two types in $\mathcal{T}$, then the most specific common supertype between T_1 and T_2, denoted as $T_1 \sqcup T_2$, is defined as follows:*

- *$T_1 \sqcup T_2 = T_1$ whenever $T_1 = T_2$;*
- *$T_1 \sqcup T_2 = type(T_1) \sqcup type(T_2)$ whenever $T_1, T_2 \in \mathcal{TN}$ and at least a type between T_1 and T_2 is not extended and unconstrained, that is either $T_1 \notin \mathcal{CT} \cup \mathcal{ET}$ or $T_2 \notin \mathcal{CT} \cup \mathcal{ET}$;*
- *$T_1 \sqcup T_2 = T$ whenever $T_1, T_2 \in \mathcal{CT} \cup \mathcal{ET}$, and $T \in \mathcal{T}$ is a supertype of T_1 and T_2, and $\forall T'$ (T' supertype of T_1 and $T_2 \Rightarrow T'$ subtype of T)[10];*
- *$T_1 \sqcup T_2 = T$ whenever $T_1, T_2 \in \mathcal{OT}$ and $T \in \mathcal{OT}$ is a superclass of T_1 and T_2, and $\forall T'$ (T' superclass of T_1 and $T_2 \Rightarrow T'$ subclass of T)[11];*

[10] We remark that T may coincide with T_1 (or with T_2), if T_1 is a supertype of T_2 (respectively, if T_2 is a supertype of T_1). We also remark that $type(T_1) = type(T_2) = type(T)$ holds.

[11] We remark that T may coincide with T_1 (or with T_2), if T_1 is a superclass of T_2 (respectively, if T_2 is a superclass of T_1).

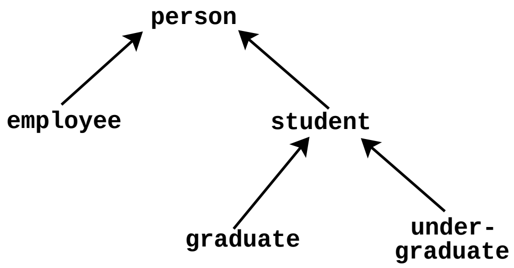

Figure 3: Inheritance hierarchy on object types of Example 8

- $T_1 \sqcup T_2 = \mathit{set\text{-}of}(T_1' \sqcup T_2')$ *whenever* $T_1 = \mathit{set\text{-}of}(T_1')$, $T_2 = \mathit{set\text{-}of}(T_2')$;
- $T_1 \sqcup T_2 = \mathit{list\text{-}of}(T_1' \sqcup T_2')$ *whenever* $T_1 = \mathit{list\text{-}of}(T_1')$, $T_2 = \mathit{list\text{-}of}(T_2')$;
- $T_1 \sqcup T_2 = \mathit{record\text{-}of}(a_1 : T_1' \sqcup T_1'', \ldots, a_n : T_n' \sqcup T_n'')$ *whenever* $T_1 = \mathit{record\text{-}of}(a_1 : T_1', \ldots, a_n : T_n')$, $T_2 = \mathit{record\text{-}of}(a_1 : T_1'', \ldots, a_n : T_n'')$;
- $T_1 \sqcup T_2$ *is undefined otherwise.* □

Example 8 *Consider a class* `person` *with a subclass* `employee` *and a subclass* `student`*. Suppose that the class* `student` *has two subclasses:* `undergraduate` *and* `graduate`*. The inheritance hierarchy is illustrated in Figure 3. Consider the constrained types* `postalcode`, `date` *and* `summerdate` *and the extended value type* `realpostalcode` *introduced in the previous subsections. Their relationships are depicted in Figure 4. Then:*

- `employee` $\sqcup$ `person` = `person`;
- `employee` $\sqcup$ `student` = `person`;
- `graduate` $\sqcup$ `undergraduate` = `student`;
- `graduate` $\sqcup$ `employee` = `person`;
- `char` $\sqcup$ `integer` *is undefined;*
- `set-of(person)` $\sqcup$ `set-of(employee)` = `set-of(person)`;
- `record-of(a:person, b:employee)` $\sqcup$ `record-of(a:employee, b:person)` =
  = `record-of(a:person, b:person)`;
- `empl-rec` $\sqcup$ `record-of(a:string, b:person)` = `record-of(a:string, b:person)`
  provided the type declaration
  `empl-rec :  record-of(a:string, b:employee)`;
- `record-of (day:integer, month:integer, year:integer)` $\sqcup$ `date` =
  = `record-of(day:integer, month:integer, year:integer)`;
- `summerdate` $\sqcup$ `date` = `date`;
- `realpostalcode` $\sqcup$ `postalcode` = `postalcode`. ◇

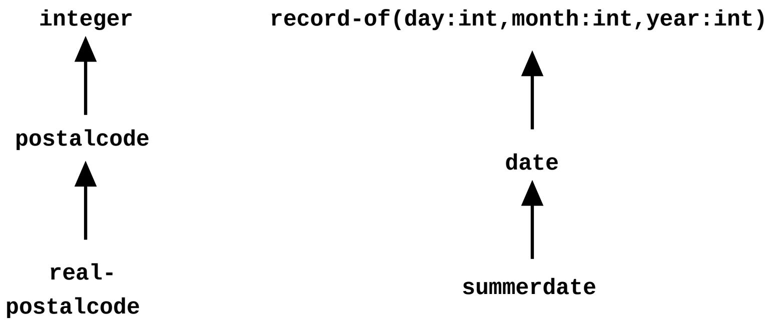

Figure 4: ISA relationships among value types of Example 8

The most specific common supertype operator $\sqcup$ defined above has the property of being an associative operator. This allows to easily generalize this notion (introduced for pairs of types) to arbitrary sets of types.

As we will see in what follows, the most specific common supertype is the least upper bound (lub) of a set of types according to the ordering on types based on inheritance relationships among types (cf. Section 5).

We remark that our lub construction is not based only upon typing aspects, rather it takes the user-defined ISA hierarchy into account. That is, in Chimera it is not possible to compute the lub of two classes having comparable underlying types, but belonging to different class hierarchies. This is quite different from most well-known systems, especially those systems developing a class taxonomy in the context of knowledge representation systems [44]. However, we think that it is not correct to allow these kinds of situations, where values which are not related to each other (only have the same underlying type) are treated as *homogeneous*. Furthermore, we must take the user-defined ISA hierarchy into account to determine which methods are applicable. Thus, the lub construction in Chimera is stricter than the lub construction in other well-known systems.

3.4 Values

In this subsection we introduce the set of Chimera legal values, and their corresponding types. First of all we introduce the set of Chimera legal values $\mathcal{V}$, and for each legal value we specify the type of which it is an instance. Note that a value may be an instance of different types.

Concerning the definition of legal values, it is important to note that in Chimera oids in $\mathcal{OI}$ are handled as values. Indeed, an object identifier $i \in \mathcal{OI}$ stands for (i.e., it is a reference to) an object o, so it is a value of an object type in $\mathcal{OT}$. More precisely, the oids of objects that belong to a class c are legal values for the object type c. Furthermore, since an object, instance of a class c, is a member of all the superclasses of c[12], for a given class we consider as legal values of the corresponding object type, all the oids of objects that belong either to the class or to one of its subclasses. For example, consider a class `person` with a subclass `employee`, the oid of an object of class `employee` is a legal value for the object type `person`.

By contrast, when considering value types, the extensions of all the predefined basic value types are assumed to be disjoint. However, when a structured value type has as a component type an object type c, the oid of an object, instance of a subclass of c, is a legal value of this component. In this sense we deal with "*heterogeneous*" structures. Consider for example the type

[12] As we discuss in the following, and according to the usual terminology, an object is an *instance* of a class if that class is the most specific one -in the inheritance hierarchy- to which the object belongs. Whenever an object o is an instance of a class c then o is also a *member* of all the superclasses of c.

`set-of(person)`, being `person` a class with a subclass `employee`. Let i_1 be the oid of an instance of class `person` and i_2 the oid of an object of class `employee`. Then $\{i_1, i_2\}$ is a legal value for the type `set-of(person)`.

Let us consider a function π assigning an extent to each extended type and to each class. Actually π is a pair of functions (π_V, π_O) such that

$$\pi_V : \mathcal{ET} \rightarrow 2^{\mathcal{V}}$$

assigns a set of values to each extended value type in $\mathcal{ET}$ (*value assignment*), and

$$\pi_O : \mathcal{CI} \rightarrow 2^{\mathcal{OI}}$$

assigns a set of object identifiers to each class in $\mathcal{CI}$ (*oid assignment*). Thus, if c is a class name in $\mathcal{CI}$, $\pi(c)$ is the set of the identifiers of objects belonging to c, both as instances and as members.

For each predefined basic value type D_i in $\mathcal{BVT}$ we postulate a non-empty extension, denoted by $dom(D_i)$. For example the extension of the basic value type `integer` is the set of integer numbers $\mathbb{Z}$, while the extension of the basic value type `bool` is the set containing the two boolean values, that is, $\{true, false\}$.

Definition 8 *(Values). The set of legal Chimera values $\mathcal{V}$ is defined as follows:*

- *for each predefined basic value type $D_i \in \mathcal{BVT}$ if $v \in dom(D_i)$ then v is a (basic) value of value type D_i (the extensions of the predefined basic value types are postulated);*
- *$i \in \mathcal{OI}$ is a (basic) value of object type c, $c \in \mathcal{OT}$, whenever $i \in \pi(c)$;*
- *null is a (basic) value of type T for each type $T \in \mathcal{T}$;*
- *let $v_1, \ldots, v_n$, $n \geq 0$ be values of types $T_1, \ldots, T_n$, respectively, such that a defined most specific common supertype T among $T_1, \ldots, T_n$ exists, then*

  $$\{v_1, \ldots, v_n\}$$

  is a (structured) value of value type set-of(T), called set value;
- *let $v_1, \ldots, v_n$, $n \geq 0$ be values of types $T_1, \ldots, T_n$, respectively, such that a defined most specific common supertype T among $T_1, \ldots, T_n$ exists, then*

  $$[v_1, \ldots, v_n]$$

  is a (structured) value of value type list-of(T), called list value;
- *if $v_1, \ldots, v_n$, $n \geq 0$ are values, and each v_i $(1 \leq i \leq n)$ has type T_i, and $a_1, \ldots, a_n \in \mathcal{AN}$ are distinct labels, then*

  $$(a_1 : v_1, \ldots, a_n : v_n)$$

  is a (structured) value of value type record-of$(a_1 : T_1, \ldots, a_n : T_n)$, called record value[13]. □

[13] We recall that for records we have the permutability of fields, that is, a value $(a_1 : v_1, \ldots, a_n : v_n)$ is identical to a value $(a_{j(1)} : v_{j(1)}, \ldots, a_{j(n)} : v_{j(n)})$ with $j : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ bijective function.

Example 9 *Provided that* i_1, i_2, i_3 *are oids in* $\mathcal{OI}$, *such that* i_1 *belongs to the extent of class* `department` *and* i_2, i_3 *belongs to the extent of class* `person`, *examples of Chimera values are the following.*

- `1, 27, 342` *are values of type* `integer`;
- $true$ *and* $false$ *are values of type* `boolean`;
- `{'a', 'k', 'z'}` *is a value of type* `set-of(character)`;
- $[true, false, false, true]$ *is a value of type* `list-of(boolean)`;
- `(re: 8.24, im: 2.18)` *is a value of type* `record-of(re:real, im:real)`;
- `{ (day:8, mont:10, year:1969), (day:5, mont:10, year:1988) }` *is a value of type* `set-of(record-of(day:integer, month:integer, year:integer))`;
- `(name:'john', dep:` i_1`)` *is a value of type* `record-of(name:string, dep:department)`;
- $\{i_2, i_3\}$ *is a value of type* `set-of(person)`. ◇

According to the above definition, and similarly to most object-oriented data models, oids are values. Indeed they are the values able to uniquely identify objects. References among objects and object sharing are supported by object identifiers. The state of an object, which is a value, may indeed contain object identifiers, representing references to other objects, as components.

In order to ensure the correctness of the database, oids must be correctly typed. Indeed, if an oid is the value of an object attribute, the oid must meet the type requirements expressed by the attribute domain. Therefore, the oid must denote an object member of the class specified as attribute domain. By contrast, oids, unlike other values, cannot be directly manipulated by the users. This is the reason why oids are not included in terms (see Definition 33). Indeed oids cannot appear in expressions written by the user, that is constraints, queries or updates, because they cannot be explicitly used. Rather, only object variables, including special variables like *self*, may appear in constraints, queries or updates. Therefore, no need arises of typing an oid when typing an expression (see Section 6.1). Oids can only be bound to variables through queries or retrieved by path expressions through navigation.

Finally note that in the above definition, the condition is imposed that an oid i is a legal value for an object type c, only if i belongs to the extent of c. However, due to Invariant 1 stated in Section 4.2, if an oid $i \in \mathcal{OI}$ identifies an object belonging to a class c, that is, if i belongs to the extent of c, the state of the object identified by i is a legal value of the type which describes the structural component of class c. This condition is formally stated as follows.

Let $stype$ be a function, defined as $stype : \mathcal{CI} \rightarrow \mathcal{VT}$, that given a class name returns the type of the states of class instances[14]. Let $value$ be a function, defined as $value : \mathcal{OI} \rightarrow \mathcal{V}$, that given an oid returns the state (that is, a structured value) of the object identified by the oid[15]. Then,

$$\forall\ i \in \mathcal{OI}, i \in \pi(c) \Rightarrow value(i) \text{ is a legal value for } stype(c).$$

Therefore, the membership to the extent of a class ensures the structural consistency of the object with respect to the class.

Definition 8, besides defining the set of Chimera legal values, implicitly states typing rules for values. The following definition explicitly states the typing rules implicit in Definition 8.

[14] We formally define this function in section 4.1 after having introduced the definition of classes.

[15] We formally define this function in section 4.2 after having introduced the definition of objects.

Definition 9 *(Typing Rules for Values). The Chimera typing rules for values are the following.*

$$\frac{}{null : T} \quad T \in \mathcal{T}$$

$$\frac{v \in dom(D_i)}{v : D_i} \quad D_i \in \mathcal{BVT}$$

$$\frac{i \in \pi(c)}{i : c} \quad i \in \mathcal{OI}, c \in \mathcal{CI}$$

$$\frac{v_i : T_i \quad (1 \leq i \leq n)}{\{v_1, \ldots, v_n\} : set\text{-}of(T)} \quad T = \bigsqcup_{i=1}^{n} T_i$$

$$\frac{v_i : T_i \quad (1 \leq i \leq n)}{[v_1, \ldots, v_n] : list\text{-}of(T)} \quad T = \bigsqcup_{i=1}^{n} T_i$$

$$\frac{v_i : T_i \quad (1 \leq i \leq n)}{(a_1 : v_1, \ldots, a_n : v_n) : record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)} \quad a_1, \ldots, a_n \in \mathcal{AN}$$

□

Note that the above typing rules do not determine a unique type for each value. In Section 6.1 we deal with typing of Chimera expressions, giving typing rules for Chimera terms and addressing the problem of assigning a unique type to an object in order to correctly handle late binding at run-time.

The above typing rules are used to check whether a database state is structurally consistent, that is, if the state of each object meets the requirements of the structural part of its class definition. In case of a named type as domain, the compatibility of the value is checked first of all with respect to the value type describing its structure; then, in case of constrained types the constraint is checked, while in case of extended types the membership to the explicit extent is checked. These concepts are formalized by Definition 10 which exactly defines the set of legal values of each type.

We define now the extension, that is, the set of legal values, of each type. The extension of a type is used, in our model, to represent the denotational semantics of the type, that is, the set of values that may be legal instances of the type. The extension of a type depends on the explicit extent for such types, like extended value types and classes, that have one. To model these extents we consider function $\pi = (\pi_V, \pi_O)$ introduced above. To make explicit that the interpretation of types can only be given under an assignment π, we denote with $[\![T]\!]_\pi$ the extension of the type T under the assignment π.

Definition 10 *(Type Interpretation).* $[\![T]\!]_\pi$ *denotes the extension of the type* T *under the assignment* π*, defined as follows:*

- $null \in [\![T]\!]_\pi$, $\forall T \in \mathcal{T}$;
- $[\![D_i]\!]_\pi = dom(D_i)$, $\forall D_i \in \mathcal{BVT}$ *(*$dom(D_i)$ *is postulated);*
- *if* $c \in \mathcal{OT}$, $[\![c]\!]_\pi = \pi(c)$;
- $[\![set\text{-}of(T)]\!]_\pi = 2^{[\![T]\!]_\pi}$, *where* 2^S *denotes the power set of the set* S;
- $[\![list\text{-}of(T)]\!]_\pi = \{[v_1, \ldots, v_n] \mid n \geq 0, v_i \in [\![T]\!]_\pi \quad \forall i, 1 \leq i \leq n\}$;

- $[\![record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)]\!]_\pi = \{(a_1 : v_1, \ldots, a_n : v_n) \mid a_i \in \mathcal{AN}, v_i \in [\![T_i]\!]_\pi \quad \forall i, 1 \leq i \leq n\}$ [16];
- $[\![Tn]\!]_\pi = [\![type(Tn)]\!]_\pi$ *if* $Tn \in \mathcal{TN} \setminus (\mathcal{CT} \cup \mathcal{ET})$;
- $[\![Tn]\!]_\pi = \{v \mid v \in [\![struct(Tn)]\!]_\pi,\ v$ *meets*[17] $constr(Tn)\}$ *if* $Tn \in \mathcal{CT}$;
- *if* $ET \in \mathcal{ET}$, $[\![ET]\!]_\pi = \pi(ET)$.

For a correct instantiation of an object type, the following constraint has to be verified

$$\{value(i) \mid i \in \pi(c)\} \subseteq [\![stype(c)]\!]_\pi \cup \bigcup_{c' \ subclasses \ of \ c} [\![stype(c')]\!]_\pi{}^{18}.$$

For a correct instantiation of an extended value type ET, we require that

$$\pi(ET) \subseteq [\![struct(ET)]\!]_\pi. \qquad \square$$

Informally, the extensions of predefined basic value types are postulated, the extension of classes and of extended types is defined as their explicit extent, while the extension of structured types is defined quite intuitively in terms of the extension of the component types. The extension of a constrained value type is the set of the legal values for the type which describes the structure restricted to those that meet the population constraint. In fact some elements belonging to the extension of the type describing the structure may not meet the formula. In addition, we impose some restrictions on value and object assignments, that is, the values (respectively, the objects) must be compatible with the structure of the extended type (respectively, the class).

Lemma 1 *If $T = T_1 \sqcup T_2$ then $[\![T_1]\!] \subseteq [\![T]\!]$ and $[\![T_2]\!] \subseteq [\![T]\!]$.*

Theorem 1 *(Soundness of typing rules for values). Let T be the type deduced for a value v according to rules in Definition 9, then $v \in [\![T]\!]$.*

The previous theorem states the soundness of our typing rules. Our typing rules for values are not complete. More precisely, they are complete with respect to $\mathcal{T} \setminus \mathcal{TN}$. That is, we never deduce for a value a named type. The result that holds is that if $v \in [\![T]\!]$ and $T \in \mathcal{T} \setminus \mathcal{TN}$, then according to typing rules in Definition 9 we deduce type T for v. By contrast, the completeness with respect to the set of all types $\mathcal{T}$ does not hold. As an example we may consider the named type `pos`, defined as the set of natural numbers. According to our typing rules, it does not exist a value v for which we deduce type `pos`, whereas, obviously, $[\![\texttt{pos}]\!] \neq \emptyset$. To get a complete type system, we must add rules that allow to deduce named types for values, but in such a way we may deduce several types for a value, because whenever a named type is deduced also the corresponding structural type is deduced, and a most specific type among the deduced ones does not exist (they are equivalent in that they have the same extension).

Our solution is to keep an incomplete (with respect to named types) type system, and, when the domain for an attribute is a named type Tn, we check that the value for the attribute is a legal value for *type*(Tn), and, as an additional constraint (that is, not related to typing), we require that the value belongs to the type interpretation $[\![T]\!]$.

[16] Note that lists are a particular case of records in which labels are the positions in the list. So we may have stated $[\![list\text{-}of(T)]\!]_\pi = \{(1 : v_1, \ldots, n : v_n) \mid v_i \in [\![T]\!]_\pi \quad \forall i, 1 \leq i \leq n\}$.

[17] Since $constr(Tn)$ is expressed in denial form by a set of rules R, v meets $constr(Tn)$ if v falsifies the bodies of all the rules in R.

[18] Equivalently, if we make use of the function ISA defined in Section 5 we can state $\{value(i) \mid i \in \pi(c)\} \subseteq [\![stype(c)]\!]_\pi \cup \bigcup_{c' \text{ s.t. } c \in ISA(c')} [\![stype(c')]\!]_\pi$.

4 Classes and Objects

4.1 Classes

The signature of a class provides all the information for the use of the class and its instances. For each class, we maintain structural, behavioral and constraint information. The signature must also specify whether the class is explicitly populated or derived. For the structural part of a class, the signature must contain name and domain of each attribute, and whether the attribute is extensional or derived. For the behavioral part of a class the signature must contain name, input and output parameters of each method. The signature must contain similar information for the structure and behavior of the class itself, that is, information about class-attributes and class-methods. The signature must finally contain for each class the signature of constraints, both those related to class instances and those related to the class itself. In this work we do not consider the trigger component of a class.

In the following we associate each class both with its proper extent, that is the set of objects belonging to the class as instances, and with its global extent, that is the set of objects belonging to the class as members. Note that the global extent can be derived as the union of the global extents of all the subclasses of the considered one. Proper and global extents coincide for classes with no subclasses.

To provide a uniform modeling of instance features and class features we introduce the notion of *metaclass*. A metaclass is a special class, whose unique instance is a class. By introducing this concept we model in a uniform way attributes and c-attributes, operations and c-operations, constraints and c-constraints. A class is seen as an instance of the corresponding metaclass exactly in the same way an object is seen as an instance of a class.

Definition 11 *(Class Signature). A class C is a tuple*

$$C = (id, pop, struct, beh, constr, state, mc)$$

where

- $id \in \mathcal{CI}$ *is the class identifier;*
- $pop \in \{ext, der\}$ *indicates whether the class is derived;*
- *struct contains the information about the structure of the instances of the class, and it is a set, containing an item for each attribute of the class, of triples*

  $$(a\_name, a\_dom, a\_type)$$

  where

  - $a\_name \in \mathcal{AN}$ *is the attribute name,*
  - $a\_dom \in \mathcal{T}$ *is the attribute domain,*
  - $a\_type \in \{ext, der\}$ *is the attribute type, that is, whether it is extensional or derived;*

- *beh contains the information about the behavior of the instances of the class, and it is a set, containing an item for each method of the class, of pairs*

  $$(op\_name, op\_sign)$$

where

- $op\_name \in \mathcal{MN}$ *is the method name,*
- op_sign *is the signature of the method, expressed as*

$$T_1 \times \ldots \times T_k \rightarrow T$$

with $T_1, \ldots, T_k$ *and* T *types in* $\mathcal{T}$, *representing, respectively, domains of input parameters and the output one of the method*[19]*;*

- *constr contains the information about the constraints on the instances of the class, and it is a set, containing an item for each constraint of the class, of pairs*

$$(con\_name, con\_sign)$$

where

- $con\_name \in \Pi$ *is the constraint name,*
- con_sign *is the signature of the constraint, expressed as*

$$T_1 \times \ldots \times T_k$$

with $T_1, \ldots, T_k$ *types in* $\mathcal{T}$, *representing domains of the output parameter of the constraint*[20]*;*

- $state \in \mathcal{V}$ *is a value containing the values for the class attributes plus two set values, one containing all the objects belonging to* C *as instances and one containing all the objects belonging to* C *as members; it is a record value of the form*

$$(a_1 : v_1, \ldots, a_n : v_n, extent : E, proper\text{-}extent : PE)$$

where $a_1, \ldots, a_n$ *are the attribute names of the c-attributes of* C*;* E *and* PE *are the sets of identifiers of objects belonging to class* C*;*

- *mc is the identifier of the metaclass corresponding to class* C, *that is, the class of which* C *is instance.* □

The attributes *extent* and *proper-extent* in the class state, denote the global and the proper extent of the class, respectively; therefore $PE \subseteq E$ ($E = PE$ if C is a leaf class in the inheritance hierarchy). The oid assignment $\pi_O : \mathcal{CI} \rightarrow 2^{\mathcal{OI}}$, introduced in Section 3.1, is such that, for each class name c, $\pi_O(c) = C.state.extent$, where C is the (unique) class such that $C.id = c$ (that is, the class identified by c).

Example 10 *An example of Chimera class is the following, where* S *and* S' *denote two oid sets.*

$id =$ `person`
$pop = ext$
$struct = \{$ (`name`, `string`, *ext*), (`vatCode`, `string`, *ext*), (`birthday`, `date`, *ext*),
(`income`, `integer`, *ext*), (`age`, `integer`, *der*) $\}$
$beh = \{$ (`changeIncome` : `integer` $\rightarrow$ `person`) $\}$
$constr = \{$ (`tooLowIncome` : `string`) $\}$
$state =$ (`averageAge` : `35`, `lifeExpentancy` : `80`, $extent : S, proper\text{-}extent : S'$)
$mc =$ `m-person`. ◇

[19] Note that we have made the assumption that the method always has a result parameter. In case of methods without result parameter, we may suppose that the method returns the object on which it has been executed. In such a case the domain of output parameter T will be the class c on which the method is defined.

[20] Note that the input parameter of the constraint is always an object of the class on which the method is defined.

The identifier of a class C denotes the object type corresponding to C. Such object type is the type of the identifiers of the objects of class C. A value type is implicitly associated with each class, representing the type of values that constitute the state of the class instances. Indeed, if a class C has as *struct* the set $\{(a_1, T_1, at_1), \ldots, (a_n, T_n, at_n)\}$, then each object instance of C must have as state a value of (record) type

$$record\text{-}of(a_1 : T_1, \ldots, a_n : T_n).$$

This type, which describes the structure of the objects of the class, is the *structural type* of the class, and it is denoted by $stype(C)$. The function $stype : \mathcal{CI} \rightarrow \mathcal{VT}$, informally introduced in Section 3.4, is simply the function that given an identifier $c \in \mathcal{CI}$ returns $t \in \mathcal{VT}$ obtained as $stype(C)$ where C is the class identified by c, that is, the (unique) class such that $C.id = c$. For example, referring to the class of Example 10, $stype(\texttt{person})$ is
$(\texttt{name} : \texttt{string}, \texttt{vatCode} : \texttt{string}, \texttt{birthday} : \texttt{date}, \texttt{income} : \texttt{integer}, \texttt{age} : \texttt{integer})$.

The *definitional component* of a class $C = (id, pop, struct, beh, constr, state, mc)$ is

$$(id, pop, struct, beh, constr, mc)$$

that is, its state-independent components.

For each class C, a metaclass MC is introduced, defined as follows. Each metaclass has a unique instance, that is, the class to which it corresponds. We remark that metaclasses in Chimera are different from metaclasses in other object-oriented models and languages, such as Smalltalk, in that they are not classes, and, thus, they cannot be instantiated multiple times. Note, however, that the notion of metaclass supported could be easily generalized by enriching the language with primitives for creating metaclasses and instantiating them.

Definition 12 *(Metaclass). A metaclass MC corresponding to a class C is a tuple*

$$MC = (id, struct, beh, constr)$$

where:

- *$id \in \mathcal{CI}$ is the identifier of the metaclass;*
- *struct contains the information about the structure of the class C, that is, its class attributes, and it is exactly like struct in Definition 11, considering class attributes instead of the instance ones, except for containing two additional extensional attributes: extent with domain $set\text{-}of(c)$ and proper-ext with domain $set\text{-}of(c)$*[21]*;*
- *beh contains the information about the behavior of the class C, that is, its class methods, and it is exactly like beh in Definition 11, considering class operations instead of instance ones;*
- *constr contains the information about the constraints of the class C, that is, its class constraints, and it is exactly like constr in Definition 11, considering class constraints instead of instance ones.* □

[21]We denote with c the identifier of the class C ($C.id$),that is, its corresponding object type.

Example 11 *The metaclass corresponding to class* `person` *of Example 10 is the following.*

id = `m-person`
$struct$ = { (`averageAge, integer`, *ext*), (`lifeExpentancy, integer`, *ext*),
    (*extent* : `set-of(person)`, *ext*), (*proper-extent* : `set-of(person)`, *ext*) }
beh = { ($\texttt{changeLifeExpentancy} : \texttt{integer} \rightarrow \texttt{m} - \texttt{person}$) }
$constr$ = { (`invalidLifeExpantancy : integer`) }. ◇

Finally, note that Chimera does not directly support the notion of abstract class. However, abstract classes can be modeled in Chimera by defining a class for which an integrity constraint is defined stating that the proper extent of the class must be empty. With an abstract class c, with proper subclasses $c_1, \ldots, c_n$ the following integrity constraint will be associated:

$$\texttt{non\_abstract(X)} \leftarrow \texttt{c(X)}, \texttt{not X in c}_1, \ldots, \texttt{not X in c}_\texttt{n}$$

This approach, however, requires that all the subclasses $c_1, \ldots, c_n$ of c are known at the time c is defined.

4.1.1 Class Implementation

In addition to a signature, classes have an implementation. Roughly speaking, the signature of a class is the definition of all the names and domains associated with that class, whereas the implementation provides the specification of the meaning of each concept associated with a class. We do not consider here active rules, thus our implementations are expressed by means of deductive (passive) rules or, in case of method implementation, of update operations constrained by a declarative formula. In a Chimera class implementation:

- (derived) attributes and c-attributes are implemented by means of deductive rules specifying the computation of values by means of a declarative expression;
- constraints and c-constraints are implemented by means of deductive rules associating with a parameterized constraint name a condition that should not hold in any state of the database;
- the implementation of an operation or c-operation is an expression of the form

  op-name : *condition* $\rightarrow$ *op-code*

  where the condition is any declarative expression of Chimera, specifying a declarative control upon operation execution, while the operation code is a sequence of update primitives, object creation and deletion, object migration from one class to another, state changes, and extended value type modifications[22].

Apart from implementing individual concepts such as attributes or constraints by means of rules, the extent of a class may be defined by means of deductive rules as well, that is, the class may be populated by means of rules referring to other classes from which the objects of the newly created class are chosen. Population rules as well as implementations of attributes, constraints, operations (both at class and instance levels) constitute the class implementation.

[22] Actually, in Chimera the operation implementation may be defined in an external programming language, but we do not consider here this case because it heavily depends on the external language which is used. Thus, we consider here only implementations expressed in the Chimera language itself.

Definition 13 *(Class Implementation). Given a Chimera class signature*

$C = (id, pop, struct, beh, constr, state, mc)$

an implementation for C consists of a set of deductive rules, specifying

- *a population implementation, if $pop = der$;*
- *an attribute implementation for each derived attribute in struct;*
- *a constraint implementation for each constraint in constr;*
- *an operation implementation for each operation in beh.* □

Note that c-attribute, c-constraint and c-operation implementations constitute the metaclass implementation.

We impose some conditions to ensure that a class implementation is consistent with the related class signature. Intuitively, an implementation for each feature specified in the signature must be provided. In addition, type compatibility must be ensured between the types defined in the signature and those returned by the implementation. We deal with these issues in Section 6.1.2, after having introduced declarative expressions and their typing.

4.2 Objects

Definition 14 *(Objects). An object is a triple*

$o = (i, v, c)$

where

- *$i \in \mathcal{OI}$ is the object identifier of o;*
- *$v \in \mathcal{V}$ is a value, called state of o;*
- *$c \in \mathcal{CI}$ is the most specific class to which o belongs, that is, being C the class such that $i \in C.state.proper\text{-}extent$, $C.id = c$.* □

Example 12 *An example of Chimera object is the following, provided that $i_1 \in \mathcal{OI}$ and* `person` $\in \mathcal{CI}$.

$(i_1,($ `name: 'john smith'`, `vatCode: '666FF'`, `birthday: (day:8, month:10, year:1969)`, `income: 5000`, `age: 25` $)$, `person` $)$ ◇

Note that the state v of an object is a value of a record value type. That is, v has the form $(a_1 : v_1, \ldots, a_n : v_n)$ with $a_1, \ldots, a_n \in \mathcal{AN}$. The function $value : \mathcal{OI} \to \mathcal{V}$ that we have informally introduced in Section 3.4, for each $i \in \mathcal{OI}$ simply returns $v \in \mathcal{V}$ such that $o = (i, v, c)$ is an object in the database. Referring to the example above

$value(i_1) = ($ `name: 'john smith'`, `vatCode: '666FF'`,
`birthday: (day:8, month:10, year:1969)`, `income: 5000`, `age: 25` $)$

An object is an *instance* of a class if this class is the most specific class[23] in the hierarchy to which the object belongs. Whenever an object o is an instance of a class C then o is also a member of all the superclasses of C. The most specific class to which an object belongs is not necessarily the one in which the object has been created (the first argument of the `create` operation). This is due to the following reasons:

1. the object may have migrated to another class, by an explicit migration operation (`specialize` or `generalize`);
2. the object may have been inserted in a more specific (derived) class, whose population predicate is satisfied by the object.

In Chimera, an object does not necessarily belong to a unique most specific class. This is due to two reasons. First of all, an object may be created as an instance of a class and then specialized in two different subclasses of the original class. Second, if an object is explicitly inserted in a class, which has two derived subclasses whose population predicates are not disjoint, and the object meets both the predicates, then the object is an instance of both these subclasses, that are not related by the subclass relationship. However, when considering objects with several most specialized classes new problems arise, because an object does no more have a single most specific class, rather it has a set of them. In such a case, an object takes the union of the features of all the classes to which it belongs. Conflicts among different definitions may however arise. Therefore, the need arises to assign a "preferred" class to each object; for example to choose which implementation to use for methods with the same name and different implementations. Thus, we leave this issue, which has been dealt with in [10], out from this specification of the model. We only mention that, to ensure that an object does not have a set of most specific classes, such insertions and specializations that would add a second most specific class to the object must be disallowed. For example, if an object inserted in a class would be inserted in two different derived subclasses of the class, not related in the ISA hierarchy, being the population predicates of both the subclasses satisfied by the inserted object, then the insertion must be rejected. Note that a static analysis of population definition to ensure the disjointness of the set of objects that would satisfy the predicates, is not feasible. Similarly, a specialization towards a class which is not a subclass of the current most specific class of an object, must be disallowed.

To unify the notion of object and class we can use a unique definition of *instance*, defined as a 7-tuple with the components of Definition 11. Objects may then be seen as particular instances, with empty *struct* and *beh*, because they cannot be further instantiated. Moreover they are instances of classes, whereas classes are instances of metaclasses, which are not instances (in fact they do not have a state and a "class" to which belong). Along this way, we may consider a unique kind of Chimera "entity", being an object an entity with a state but without specification part, a class an entity with both a state and a specification part, and a metaclass an entity without state, but only with a specification component.

As we will see in Section 5.4 each object must be a consistent instance of all the classes to which it belongs. Therefore an object $o = (i, v, c)$ must be a consistent instance of class c[24]. We distinguish two kinds of consistency:

- *structural consistency* if the instance respects the structure of the class in which it is defined, that is, if the object state is a (record) value whose type is the type of the structural component of the class;

[23] This notion will be formally defined in Section 5.

[24] This ensures also the consistency with respect to all the superclasses of class c.

- *constraint consistency* if the instance satisfies the constraints of the class in which it is defined.

Definition 15 *(Structural Consistency). An object $o = (i, v, c)$ is a structurally consistent instance of a class c' if v is a legal (record) value for the type $stype(c')$.* □

The above definition states that the state of each object must contain a value for each attribute of the class, and that this value must be of the correct type, that is, it must meet the domain specification.

Definition 16 *(Constraint Consistency). An object $o = (i, v, c)$ is a constraint consistent instance of a class c' if o falsifies all the bodies of rules implementing the constraints in the constr component of C, where $C.id$ is c.* □

Definition 17 *(Consistency). An object $o = (i, v, c)$ is a consistent instance of a class c' if o is both a structural and constraint consistent instance of c'.* □

Invariant 1 *An object $o = (i, v, c)$ must be a consistent instance of class c.*

Example 13 *Consider the object of Example 12 and class* `person` *of Example 10.*

$value(i_1) = ($ `name: 'john smith'`, `vatCode: '666FF'`,
`birthday: (day:8, month:10, year:1969)`, `income: 5000`, `age: 25` $)$

is a legal value of $stype($`person`*$) =$*

$(\mathtt{name} : \mathtt{string}, \mathtt{vatCode} : \mathtt{string}, \mathtt{birthday} : \mathtt{date}, \mathtt{income} : \mathtt{integer}, \mathtt{age} : \mathtt{integer})$,

thus structural consistency holds.

Furthermore, if the implementation for the constraint `tooLowIncome` *is specified by the following rule*

$$\mathtt{tooLowIncome(N)} \leftarrow \mathtt{Self.income} < \mathtt{5000}, \mathtt{N} = \mathtt{Self.name}$$

the atom $\mathtt{Self.income} < \mathtt{5000}$ *is falsified by the object (since* $\mathtt{Self.income} = \mathtt{5000}$*), thus constraint consistency holds as well.* ◇

An object (i, v, c) is said to *depend on* (or to *refer to*) an object (i', v', c') if i' appears in v. A finite set of objects OBJ is consistent if the set is closed under the "depend-on" relation, that is, if for each object in the set all objects referred by it belong to the set. This property is also known as *referential integrity* of the set. Intuitively speaking, this notion states that because identifiers are pointers to objects there must be no *dangling pointers* in the set. In addition, for a set of objects to be consistent, the property of oid-uniqueness must be ensured. The following definition formalizes these concepts.

Definition 18 *(Consistent set of objects). A (finite) set of objects OBJ is consistent if and only if*

1. OID-UNIQUENESS

   $\forall o_1, o_2 \in OBJ$, *if* $o_1.i = o_2.i$, *then* $o_1.v = o_2.v$ *and* $o_1.c = o_2.c$;

2. REFERENTIAL INTEGRITY

$\forall o \in OBJ,\ ref(o) \subseteq \{i \mid \hat{o} = (i, v, c), \hat{o} \in OBJ\}$. □

where, given an object $o = (i, v, c)$, $ref(o)$ denotes the set of identifiers in $\mathcal{OI}$ appearing in v.

Example 14 *Consider the following objects*

- $o_1 = (i_1,$ `(name:'Alan Ford', children:` $\{i_3, i_4\}$`), person)`;
- $o_2 = (i_2,$ `(name:'computer science', chief:` i_1`), department)`;
- $o_3 = (i_3,$ `(name:'Susan Ford', children:{ }), person)`;
- $o_4 = (i_4,$ `(name:'Bill Ford', children: { }, department :` i_2`), student)`.

The set $OBJ = \{o_1, o_2, o_3, o_4\}$ is a consistent set of objects. By contrast, $OBJ' = \{o_1, o_2\}$ is not a consistent set of objects, because the references i_3 and i_4 are "dangling". ◇

Referential integrity is preserved in Chimera by allowing the deletion of referenced objects and removing all the references to deleted objects. This removal is performed setting to *null* each reference to the object, if it were the value for an atomic attribute. By contrast, if the object identifier were a component of a set or list-valued attribute, then the reference is removed from the set/list.

We consider now the notion of object equality. In value-based systems there is no need to distinguish between identical objects and equal objects, since the two notions are the same. By contrast, object-oriented systems need to distinguish them as there is a sharp distinction between values and objects. Classically [32], there are two fundamentally different notions of equality that are supported by object-oriented systems: *equality by identity* (meaning that the two denoted objects are the same object) and *equality by value* (meaning that the two denoted objects have the same attribute values, but not necessary the same identifier). The latter form of equality can be further refined in two ways: *shallow equality* considers the equality of all direct attributes (possibly represented by means of object identifiers) of an object, while *deep equality* also considers the equality of the attributes of objects which are recursively reached by means of oid references. The following definitions formally state these concepts.

Definition 19 *(Equality by Identity). Two objects $o_1 = (i_1, v_1, c_1)$ and $o_2 = (i_2, v_2, c_2)$ are equal by identity (denoted by $o_1 = o_2$) if and only if $i_1 = i_2$ (and obviously $v_1 = v_2$, and $c_1 = c_2$ because of oid-uniqueness).* □

Definition 20 *(Shallow Value Equality). Two objects $o_1 = (i_1, v_1, c_1)$ and $o_2 = (i_2, v_2, c_2)$ are equal by value under shallow equality (denoted by $o_1 == o_2$) if and only if $v_1 = v_2$. This implies both the equality of the attribute values and the equality of the attribute names.* □

Note that shallow value equality does not even require that the two objects are instances of classes related in the inheritance hierarchy, provided that the objects have the same attributes.

Example 15 *Consider a database containing a class* `person` *and a class* `company`*, obviously not related in the inheritance hierarchy. Let*

stype(`person`) = *stype*(`company`) = `record-of(name:string, address:string)`.

Consider the objects

- $o_1 = (i_1, (\texttt{name} : \text{`}\texttt{Ford}', \texttt{address} : \text{`}\texttt{Austin}'), \texttt{person})$;
- $o_2 = (i_2, (\texttt{name} : \text{`}\texttt{Ford}', \texttt{address} : \text{`}\texttt{Austin}'), \texttt{company})$.

o_1 and o_2 are equal by value under shallow equality. ◇

Before introducing deep value equality, we introduce the notion of span graph of an object. Intuitively the span graph of an object is the graph obtained by recursively replacing each oid-reference in the object value with the state of the object referenced by it. Note that this operation results in a graph, instead of a tree, because of the possibility of circular references among objects.

We introduce an operator $\hat{}: \mathcal{V} \to \mathcal{V}$, which given a value cuts off the fields of the value which are oids [25]. The operator is inductively defined in the following way. Given a value $v \in \mathcal{V}$, $\hat{v}$ is

- $\hat{null} = null$;
- if $v \in dom(D_i)$, with $D_i \in \mathcal{BVT}$, that is, v is a basic value, then $\hat{v} = v$;
- if $v = \{v_1, \ldots, v_n\}$ then $\hat{v} = \{\hat{v_{i_1}}, \ldots, \hat{v_{i_m}}\}$ where $m \leq n$, and $\{v_{i_1}, \ldots, v_{i_m}\} \subseteq \{v_1, \ldots, v_n\}$ is obtained from $\{v_1, \ldots, v_n\}$ removing those v_i's for which $\hat{v_i}$ is undefined;
- if $v = (a_1 : v_1, \ldots, a_n : v_n)$ then $\hat{v} = (a_{i_1} : \hat{v_{i_1}}, \ldots, a_{i_m} : \hat{v_{i_m}})$ where $m \leq n$, and $\{(a_{i_1}, v_{i_1}), \ldots, (a_{i_m}, v_{i_m})\} \subseteq \{(a_1, v_1), \ldots, (a_n, v_n)\}$ is obtained from $\{(a_1, v_1), \ldots, (a_n, v_n)\}$ removing those pairs (a_i, v_i) for which $\hat{v_i}$ is undefined[26];
- $\hat{v}$ is undefined otherwise.

Note that according to this definition $\hat{\imath}$, $i \in \mathcal{OI}$, is undefined.

Definition 21 *(Span Graph)*[27]*. The span graph of an object $o = (i, v, c)$ (denoted as $span\_graph(o)$), is a directed graph (VE, E) such that*

- *vertices are values in $\mathcal{V}$, not containing identifiers in $\mathcal{OI}$ as components;*
- *edges are labeled by attribute names in $\mathcal{AN}$.*

The graph is defined as follows:

- *$\hat{v}$ is a vertex in VE;*
- *if a vertex $\hat{v}^*$ exists in VE, $\hat{v}^*$ has been obtained from a value v^*, and $i^* \in \mathcal{OI}$ appears in v^* as the value for an attribute a^*[28], then an edge from $\hat{v}^*$ to $\hat{ve}$ with label a^* is added to the graph. $\hat{ve}$ is obtained from the value ve, where ve is $o^*.v$ and $o^*.i = i^*$. If such a vertex is not in VE, then the vertex is added to VE.* □

Example 16 *Consider the following objects:*

- $o_1 = (i_1,$ `(name:'Alan Ford', children:` $\{i_3, i_4\}), \texttt{person})$;

[25] This is due to the fact that when considering deep value equality, oids are not considered in the test, rather they are expanded into their corresponding values.

[26] The case $v = [v_1, \ldots, v_n]$ is handled similarly, regarding v as $(1 : v_1, \ldots, n : v_n)$, that is, as a record value with natural labels.

[27] This definition assumes that list values are seen as record values with natural labels.

[28] Or as a component of the set which is the value for a^*.

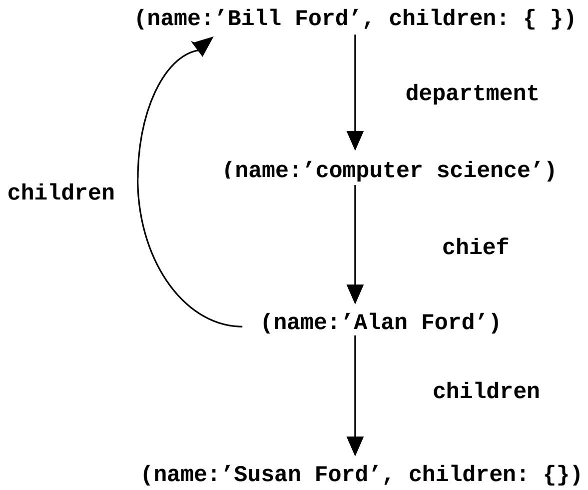

Figure 5: Span graph of object o_4 of Example 16

- $o_2 = (i_2,$ `(name:‘computer science’, chief:`i_1`),department`$)$;
- $o_3 = (i_3,$ `(name:‘Susan Ford’, children:` $\{\ \}$`),person`$)$;
- $o_4 = (i_4,$ `(name:‘Bill Ford’, children:` $\{\ \}$`,department:`i_2`),student`$)$.

The span graph of object o_4 is shown in Figure 5. ◇

We are now able to give the definition of deep value equality.

Definition 22 *(Deep Value Equality). Two objects $o_1 = (i_1, v_1, c_1)$ and $o_2 = (i_2, v_2, c_2)$ are equal by value under deep equality (denoted by $o_1 ==_d o_2$) if and only if $span\_graph(o_1) = span\_graph(o_2)$, where the equality of two graphs is the equality of their vertices and the equality of their edges.* □

Clearly equality by identity implies equality by value, and, among equalities by value, shallow equality implies deep equality. Indeed, shallow equality requires that all the objects referred by the two compared objects are exactly the same. Deep equality, by contrast, (recursively) requires the equality of the states of the objects referred by the two compared objects. Thus, because of oid uniqueness, shallow equality implies deep equality.

When we consider equalities among typed expression, we may specialize these notions of equalities to compare only the components of the states related to the type assigned to the expression. In such a way, we consider different kinds of equality. However, we do not elaborate on this issue here.

5 Inheritance

Inheritance relationships among classes are described by an ISA hierarchy established by the user. This ISA hierarchy represents which classes are subclasses of (inherit from) other classes. This information is expressed as a function

$$ISA : \mathcal{CI} \rightarrow 2^{\mathcal{CI}}$$

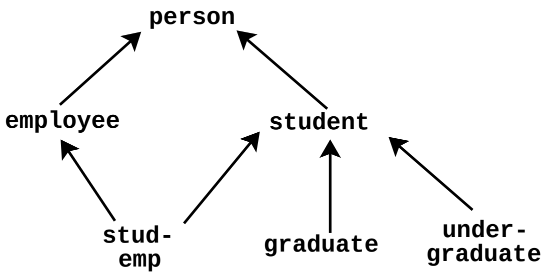

Figure 6: Inheritance hierarchy of Example 17

such that for each $c \in \mathcal{CI}$, $ISA(c)$ is the set of the (direct) superclasses of c. The function $ISA^* : \mathcal{CI} \rightarrow 2^{\mathcal{CI}}$ is the transitive closure of the subclass relationship, that is, it returns all the superclasses (also the indirect ones) of a given class.

Function ISA^* is defined as follows:

$$ISA^*(c) = \begin{cases} \emptyset & \text{if } ISA(c) = \emptyset \\ \cup_{c' \in ISA(c)} ISA^*(c') & \text{otherwise} \end{cases}$$

Example 17 *Referring to the inheritance hierarchy depicted in Figure 6 the ISA relationships are the following:*

$ISA(\texttt{person}) = \emptyset$
$ISA(\texttt{employee}) = \{\texttt{person}\}$
$ISA(\texttt{student}) = \{\texttt{person}\}$
$ISA(\texttt{stud-emp}) = \{\texttt{employee}, \texttt{student}\}$
$ISA(\texttt{graduate}) = ISA(\texttt{under-graduate}) = \{\texttt{student}\}$

$ISA^*(\texttt{person}) = ISA(\texttt{person}) = \emptyset$
$ISA^*(\texttt{employee}) = ISA(\texttt{employee}) = \{\texttt{person}\}$
$ISA^*(\texttt{student}) = ISA(\texttt{student}) = \{\texttt{person}\}$
$ISA^*(\texttt{stud-emp}) = \{\texttt{employee}, \texttt{student}, \texttt{person}\}$
$ISA^*(\texttt{graduate}) = ISA^*(\texttt{under-graduate}) = \{\texttt{student}, \texttt{person}\}$. $\diamond$

A set of conditions must be satisfied by two classes related by the ISA relationship. These conditions are related to the fact that each subclass must contain all attributes, operations, constraints (both on the class as well on the instance level) of all its superclasses. Apart from the inherited concepts, additional features can be introduced in a subclass. Inherited concepts may be redefined (overwritten) in a subclass definition under a number of restrictions. Indeed, in Chimera the redefinition of the signature of an attribute is possible by specializing, that is, refining, the domain of the attribute. The redefinition of the signature of an operation must verify the *covariance rule* for result parameters and the *contravariance rule* for the input ones. Therefore, result parameter domains may be specialized, whereas input parameter domains may be generalized, in the subclass signature of the operation. The implementation of an attribute or an operation may be redefined as well, introducing a different implementation of the respective concept, which "overrides" the inherited definition. The redefinition of derived and extensional attributes is not allowed if a derived attribute becomes extensional or vice-versa. We also require that the extent of a subclass is a subset of the extent of all its superclasses.

First of all we need to define an ordering on types, to formally define notions such as *domain refinement*. Besides the ISA ordering on classes, and thus on object types, another ordering must be considered, namely the ordering on value types due to constrained value types and to extended value types. Consider the function

$$ISA_V : \mathcal{VT} \to \mathcal{VT}$$

that, given a constrained value type or an extended value type returns the (possibly constrained) value type in terms of which it has been defined (which is its parent in the ISA hierarchy). Formally

- if $T \in \mathcal{CT}$ is a constrained type, then $ISA_V(T) = struct(T)$
- if $T \in \mathcal{ET}$ is an extended type, then $ISA_V(T) = struct(T)$
- $ISA_V(T) = \emptyset$ otherwise, that is, if $T \in \mathcal{VT} \setminus (\mathcal{CT} \cup \mathcal{ET})$

Let ISA_V^* denote the transitive closure of the ISA_V relationship. We remark that our ISA_V^* hierarchy is such that $T \in ISA_V^*(T') \Rightarrow type(T) = type(T')$.

5.1 Subtype Relationship

The subtype relationship $\leq_T$ is defined as follows. Note that the subtype relationship for basic types is the identity.

Definition 23 *(Subtypes). Given $T_1, T_2 \in \mathcal{T}$, T_2 is a subtype of T_1 (denoted as $T_2 \leq_T T_1$) if and only if one of the following conditions holds:*

- $T_1 = T_2$;
- $T_1 \in ISA^*(T_2)$;
- $T_1 \in ISA_V^*(T_2)$;
- *T_1 and T_2 are non extended unconstrained value types (that is, $T_1, T_2 \in \mathcal{VT} \setminus (\mathcal{CT} \cup \mathcal{ET})$) and $type(T_1) = type(T_2)$;*
- $T_2 = set\text{-}of(T_2')$, $T_1 = set\text{-}of(T_1')$ *and* $T_2' \leq_T T_1'$;
- $T_2 = list\text{-}of(T_2')$, $T_1 = list\text{-}of(T_1')$ *and* $T_2' \leq_T T_1'$;
- $T_1 = record\text{-}of(a_1 : T_1', \ldots, a_n : T_n')$, $T_2 = record\text{-}of(a_1 : T_1'', \ldots, a_n : T_n'')$ *and for each* i, $1 \leq i \leq n$, $T_i'' \leq_T T_i'$. $\square$

Example 18 *Consider the set of object types* { `person, student, employee, stud-emp, graduate, under-graduate` } *with the ISA hierarchy of Example 17. Consider the set of named types* { `date, summerdate, postalcode, realpostalcode, colour` } *such that*

$$ISA_V^*(\texttt{date}) = \{\texttt{record-of(day : integer, month : integer, year : integer)}\}$$
$$ISA_V^*(\texttt{summerdate}) = \{\texttt{date, record-of(day : integer, month : integer, year : integer)}\}$$
$$ISA_V^*(\texttt{colour}) = \{\texttt{string}\}$$
$$ISA_V^*(\texttt{postalcode}) = \{\texttt{integer}\}$$
$$ISA_V^*(\texttt{realpostalcode}) = \{\texttt{postalcode, integer}\}.$$

Then, the following subtype relationships hold:

`graduate` $\leq_T$ `student` $\leq_T$ `person`
list-of(`stud-emp`) $\leq_T$ *list-of*(`student`) $\leq_T$ *list-of*(`person`)
`summerdate` $\leq_T$ `date`
`date` $\leq_T$ `record-of(day:integer, month:integer, year:integer)`
`record-of(a:colour, b:postalcode)` $\leq_T$ `record-of(a:string, b:integer)`. $\diamond$

The set $\mathcal{T}$ of types with the ordering $\leq_T$ is a poset. Indeed $\leq_T$ is a partial order: it can be easily checked that it is reflexive, antisymmetric and transitive. Given the poset $(\mathcal{T}, \leq_T)$ and a set $TS \subseteq \mathcal{T}$ we may consider the upper bound of TS (the type $T \in \mathcal{T}$ such that $T_{TS} \leq_T T \ \ \forall T_{TS} \in TS$). Intuitively an upper bound of a set of classes is a class C superclass of all the classes in the set. Consider for instance the hierarchy of Example 8, depicted in Figure 3. The upper bound of the set { `employee`, `person` } is `person`, as the upper bound of the set { `employee`, `student` } and of the set { `graduate`, `employee` }. By contrast, the set { `graduate`, `undergraduate` } has two upper bounds: `student` and `person`. This can be easily generalized to structured types with object types as components.

We may consider the notion of *least upper bound* (*lub*) in our poset. An upper bound T for a set TS is the least upper bound if and only if for all upper bounds T', $T \leq_T T'$ holds. Therefore, the *lub* of a set of types is the most specific among the supertypes of the types in the set. Referring to the above example the *lub* of { `graduate`, `undergraduate` } is `student`, since it is more specific than `person`. The following theorem states that the most specific common supertype of a set of types defined in Definition 7 is the least upper bound of the set considering the poset $(\mathcal{T}, \leq_T)$.

Theorem 2 *If $T_1 \sqcup T_2 = T$, then the following conditions hold:*

- *$T_1 \leq_T T$ and $T_2 \leq_T T$ (that is, T is an upper bound);*
- *$\forall T^*$ such that $T_1 \leq_T T^*$ and $T_2 \leq_T T^*$, $T \leq_T T^*$ (that is, T is the least among the upper bounds).*

Note however that the lub does not always exist, nor it is always unique. As a simple example, the lub of two types without common supertypes does not exist (this follows immediately from Definition 7). Furthermore, it is possible that the lub is not unique. Consider for example the following situation. Let c and c' be two classes not related by the ISA hierarchy, and c_1 and c_2 be two classes both subclasses of both c and c' (this situation is illustrated in Figure 7). Then $c_1 \sqcup c_2$ would be either c or c'. We may construct a new type being the required (unique) lub, but this would imply to add a new class into the class hierarchy. Thus, we must look for a (already defined) lub class, which, indeed, may not be unique. We have stated (cf. Definition 7) that the lub only exists if it is unique, that is,

$T_1 \sqcup T_2 = T$ whenever T is a superclass of T_1 and T_2 and $\forall T'(T'$ superclass of T_1 and $T_2 \Rightarrow T'$ superclass of $T)$.

Alternatively, we may have said that the lub exists, but, to avoid a type error, the user should explicitly state what the lub must be. Suppose indeed to have an expression e_1 of type c_1 and an expression e_2 of type c_2. According to our decision, the set expression $\{e_1, e_2\}$ is not legal. Alternatively, we may allow the user to use explicit conversion function to handle these kinds of situations.

Finally, the following result holds, relating the ordering on types to type extensions defined in Subsection 3.4.

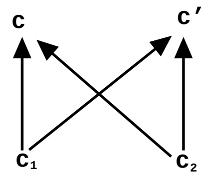

Figure 7: Inheritance hierarchy showing *mscst* non-uniqueness

Theorem 3 *If $T_1 \leq_T T_2$, then $[\![T_1]\!] \subseteq [\![T_2]\!]$ holds.*

We have no result on the converse of Theorem 3, because the ordering on type $\leq_T$ we consider keeps into account only the ISA relationships explicit in type definitions, but it does not consider those implicit in type constraints. Indeed, consider for example a constrained type `geq5` such that *struct*(`geq5`) = `integer` and *constr*(`geq5`) = {`imprgeq5(X)` $\leftarrow$ `X` $<$ `5`}, and a constrained type `geq7` such that *struct*(`geq7`) = `integer` and *constr*(`geq7`) = {`imprgeq7(X)` $\leftarrow$ `X` $<$ `7`}. Then, both `geq5` $\leq_T$ `integer` and `geq7` $\leq_T$ `integer` hold, while `geq5` and `geq7` are not related in the $\leq_T$ ordering. By contrast $[\![$`geq7`$]\!] \subseteq [\![$`geq5`$]\!]$, being each integer number greater or equal to 7 an integer number greater or equal to 5.

5.2 Signature Refinement

An ordering among signatures is now imposed in the following way. Note that we consider covariance for output parameter and contravariance for input ones. Specialization of domains of output parameters of operations can therefore be done by replacing the domain with a proper subtype (*covariance rule*). By contrast, input parameters of operations can be refined by replacing the domain with a proper supertype (*contravariance rule*).

Definition 24 *(Signature Refinement). Let $s = T_1 \times \ldots \times T_k \rightarrow T$ and $s' = T'_1 \times \ldots \times T'_k \rightarrow T'$ be two signatures. We say that s is a refinement of s', denoted by $s \leq_S s'$ if and only if for each i, $1 \leq i \leq k$, $T'_i \leq_T T_i$ and $T \leq_T T'$.* □

Example 19 *Referring to the ordering of Example 18, given s_1 and s_2 defined as follows*

s_1 = `person` $\times$ `employee` $\times$ `postalcode` $\rightarrow$ `summerdate`
s_2 = `graduate`$\times$ `stud-emp` $\times$`realpostalcode` $\rightarrow$ `date`

then $s_1 \leq_S s_2$ holds. ◇

5.3 Subclasses

Based on the ordering among types, we define an ordering on classes to ensure that the user-defined ISA hierarchy meets the compatibility conditions. Firstly, we define an ordering among metaclasses, imposing compatibility conditions in refinement of class features.

Definition 25 *Given $MC_1 = (id_1, struct_1, beh_1, constr_1)$, $MC_2 = (id_2, struct_2, beh_2, constr_2)$ metaclasses, MC_2 precedes MC_1 (denoted as $MC_2 \leq_{MC} MC_1$) if and only if all the following conditions hold:*

- $struct_1 = \{(a'_1, T'_1, at'_1), \ldots, (a'_{k_1}, T'_{k_1}, at'_{k_1})\}$
  $struct_2 = \{(a''_1, T''_1, at''_1), \ldots, (a''_{k_2}, T''_{k_2}, at''_{k_2})\}$
  and for each $i = 1 \ldots k_1$ (a''_j, T''_j, at''_j) exists, $1 \leq j \leq k_2$, such that

- $a'_i = a''_j$,
- $T''_j \leq_T T'_i$,*that is, the domain of the attribute may be refined in a subtype of the domain in the superclass,*
- $at'_i = at''_j$, *that is, the type of the attribute may not be changed from extensional to derived nor vice-versa;*

* $beh_1 = \{(m'_1, s'_1), \ldots, (m'_{h_1}, s'_{h_1})\}$
  $beh_2 = \{(m''_1, s''_1), \ldots, (m''_{h_2}, s''_{h_2})\}$
  and for each $i = 1 \ldots h_1$ (m''_j, s''_j) *exists,* $1 \leq j \leq h_2$, *such that*

  - $m'_i = m''_j$,
  - $s''_j \leq_S s'_i$, *that is, the signature of the operation is a refinement of the signature in the superclass;*

* $constr_1 = \{(con'_1, s'_1), \ldots, (con'_{h_1}, s'_{h_1})\}$
  $constr_2 = \{(con''_1, s''_1), \ldots, (con''_{h_2}, s''_{h_2})\}$
  and for each $i = 1 \ldots h_1$ (con''_j, s''_j) *exists,* $1 \leq j \leq h_2$, *such that*

  - $con'_i = con''_j$,
  - $s'_i = s''_j$, *that is, we do not consider constraint refinement*[29]. □

We are now able to introduce the ordering among classes. First of all we introduce two different orderings on classes, one considering the time-invariant component of a class (that is, its definitional component) and the other considering the time-varying one (that is, its state and extension). To express compatibility conditions we take into account both orderings.

Definition 26 *(Intensional Ordering). Given* $C_1 = (id_1, pop_1, struct_1, beh_1, constr_1, state_1, mc_1)$, *and* $C_2 = (id_2, pop_2, struct_2, beh_2, constr_2, state_2, mc_2)$ *classes,* C_2 *precedes* C_1 *in the intensional ordering (denoted as* $C_2 \leq^i_C C_1$*) if and only if all the following conditions hold:*

* $struct_1 = \{(a'_1, T'_1, at'_1), \ldots, (a'_{k_1}, T'_{k_1}, at'_{k_1})\}$
  $struct_2 = \{(a''_1, T''_1, at''_1), \ldots, (a''_{k_2}, T''_{k_2}, at''_{k_2})\}$
  and for each $i = 1 \ldots k_1$ (a''_j, T''_j, at''_j) *exists,* $1 \leq j \leq k_2$, *such that*

  - $a'_i = a''_j$,
  - $T''_j \leq_T T'_i$, *that is, the domain of the attribute may be refined in a subtype of the domain in the superclass,*
  - $at'_i = at''_j$, *that is, the type of the attribute may not be changed from extensional to derived nor vice-versa;*

* $beh_1 = \{(m'_1, s'_1), \ldots, (m'_{h_1}, s'_{h_1})\}$
  $beh_2 = \{(m''_1, s''_1), \ldots, (m''_{h_2}, s''_{h_2})\}$
  and for each $i = 1 \ldots h_1$ (m''_j, s''_j) *exists,* $1 \leq j \leq h_2$, *such that*

  - $m'_i = m''_j$,
  - $s''_j \leq_S s'_i$, *that is, the signature of the operation is a refinement of the signature in the superclass;*

[29] The issue of constraint (and trigger) refinement in Chimera is currently under investigation [11].

- $constr_1 = \{(con'_1, s'_1), \ldots, (con'_{h_1}, s'_{h_1})\}$
  $constr_2 = \{(con''_1, s''_1), \ldots, (con''_{h_2}, s''_{h_2})\}$
  and for each $i = 1 \ldots h_1$ (con''_j, s''_j) *exists,* $1 \leq j \leq h_2$*, such that*
  - $con'_i = con''_j$,
  - $s'_i = s''_j$,*that is, we do not consider constraint refinement*
- $mc_2 \leq_{MC} mc_1$*, that is, we impose the same rules for the refinement of class features, as for refinement of instance features.* □

Note that we have imposed no conditions on pop_1 and pop_2, thus an extensional class may have a derived subclass, but also the converse may hold.

Example 20 *Consider class* `person` *defined as follows*

id = `person`
pop = *ext*
struct = { *(*`name, string`*, ext), (*`birthday, date`*, ext), (*`spouse, person`*, ext),*
(`income, integer`*, ext), (*`age, integer`*, der)* }
beh = { (`changeIncome : integer` → `person`) }
constr = { `tooLowIncome : string`) }
mc = `m-person`

and the following class `employee`

id = `employee`
pop = *der*
struct = { *(*`name, string`*, ext), (*`birthday, date`*, ext), (*`spouse, employee`*, ext),*
(`income, integer`*, ext), (*`age, integer`*, der), (*`empnr, integer`*, ext),*
(`manager, employee`*, ext), (*`dependents, set-of(employee)`*, ext)* }
beh = { (`changeIncome : integer` → `employee`) }
constr = { `tooLowIncome : string`
`exceedMgrSalary : integer` }
mc = `m-employee`

Then, provided that `m-employee` $\leq_{MC}$ `m-person`*,* `employee` $\leq^i_C$ `person`*.* ◇

Definition 27 *(Extensional Ordering). Given* $C_1 = (id_1, pop_1, struct_1, beh_1, constr_1, state_1, mc_1)$*, and* $C_2 = (id_2, pop_2, struct_2, beh_2, constr_2, state_2, mc_2)$ *classes,* C_2 *precedes* C_1 *in the extensional ordering (denoted as* $C_2 \leq^e_C C_1$*) if and only if* $state_2.extent \subseteq state_1.extent$*, that is, the set objects members of the subclass is included in the set of the members of the superclass.* □

Note that, as we will see in Section 5.4, in each consistent database the ISA hierarchy must be consistent with the orderings $\leq^i_C$ and $\leq^e_C$. We thus require that the ISA specification (the subclass relationship) does not contradict the defined orderings. This consistency must be ensured at two different levels: the intensional and the extensional ones. At the intensional level (schema level) the ordering on classes imposed by the ISA hierarchy is required to be consistent with the intensional ordering on classes. Therefore, each subclass must contain all the features of the superclasses, possibly refined. At the extensional level (instance level), the ordering on classes imposed by the

ISA hierarchy is required to be consistent with the extensional ordering on classes. Intuitively an ISA relationship is well-defined if it is consistent with the compatibility conditions stated above.

In what follows, for simplicity of notation, we apply function ISA^* to classes, instead of to class identifiers. Let us denote with $\mathcal{C}$ the set of all classes. Then, $ISA^* : \mathcal{C} \rightarrow \mathcal{C}$ is simply defined as

$ISA^*(C) = \{C' \mid C'.id \in ISA^*(C.id)\}$.

This mapping is very intuitive due to the one-to-one correspondence between classes and class names.

Definition 28 *(Well-defined Inheritance Relationship). An inheritance relationship ISA is said to be intensionally-well-defined if* $\forall C_1, C_2 \in \mathcal{C}$, $C_2 \in ISA^*(C_1) \Rightarrow C_1 \leq^i_C C_2$, *that is, if whenever a class* C_2 *is a subclass of a class* C_1 *then* C_2 *precedes* C_1 *in the intensional ordering.*

An inheritance relationship ISA is said to be extensionally-well-defined if $\forall C_1, C_2 \in \mathcal{C}$, $C_2 \in ISA^*(C_1) \Rightarrow C_1 \leq^e_C C_2$, *that is, if whenever a class* C_2 *is a subclass of a class* C_1 *then* C_2 *precedes* C_1 *in the extensional ordering.*

An inheritance relationship ISA is said to be well-defined if it is both intensionally-well-defined and extensionally-well-defined, namely, if $\forall C_1, C_2 \in \mathcal{C}$, $C_2 \in ISA^*(C_1) \Rightarrow C_1 \leq^i_C C_2 \wedge C_1 \leq^e_C C_2$ *that is, if whenever a class* C_2 *is a subclass of a class* C_1 *then* C_2 *precedes* C_1 *both in the extensional and in the extensional ordering.* □

By contrast, two classes $C_1, C_2 \in \mathcal{C}$ may exist such that $C_1 \leq^i_C C_2$ and $C_1 \leq^e_C C_2$, but $C_1 \notin ISA^*(C_2)$. Consider indeed the case of a class `person` with two subclasses `male` and `student` which simply inherit structure and behavior (both at the instance and at the class level) without adding nor refining anything. The two classes have therefore identical structure and behavior. So we have `student` $\leq^i_C$ `male` (and also the vice-versa). Suppose that all students are male, then we have `student`.*extent* $\subseteq$ `male`.*extent* and therefore `student` $\leq^e_C$ `male`. In this case `student` $\leq^i_C$ `male` and `student` $\leq^e_C$ `male`, while `male` $\notin ISA^*($`student`$)$.

We are now able to formally state the notion of most specific class in a set of classes, which we have already informally introduced. The most specific classes in a set CS are those which do not have a subclass in the set.

Definition 29 *(Most Specific Class). Given a set of classes* $CS \subseteq \mathcal{C}$, *a class* C *is a most specific class in* CS *if* $\nexists\, C' \in CS$ *such that* $C \in ISA(C')$. □

Example 21 *Consider the inheritance hierarchy of Example 17. The most specific class of the set* { `person, employee, stud-emp`} *is the class* `stud-emp`. *Similarly, the most specific class of the set* { `person, student, graduate`} *is the class* `graduate`. *By contrast, a unique most specific class in the set* { `person, student, employee` } *does not exist.* ◇

We point out that in Chimera a common superclass of all the classes in $\mathcal{C}$ does not exist. Therefore the hierarchy is actually a DAG, consisting of a number of connected components whose roots are the classes without superclasses, which we call root classes.

Definition 30 *(Root Class). A class* C *is a root if* $ISA(C) = \emptyset$. □

Example 22 *Referring to the inheritance hierarchy of Example 17 the only root class is* `person`. ◇

In Chimera multiple inheritance is supported. However the constraint is imposed that for multiple inheritance a common ancestor must exist. Therefore a class C can be defined as a subclass of classes C_1 and C_2 only if $ISA^*(C_1) \cap ISA^*(C_2) \neq \emptyset$.

Furthermore, since we consider objects which are instances of a unique class, the sets of oids in different hierarchies, that is, hierarchies with different roots, are disjoint. Consider a set of root classes $C_1, \ldots, C_m$, then Ext_i, $i = 1, \ldots, m$, denotes the extension of C_i ($C_i.state.extent$) which is the extent of the entire hierarchy rooted at C_i. Note that we may think at $\mathcal{C}$ as partitioned in $\mathcal{H}_1, \ldots, \mathcal{H}_m$,that is, into m distinct hierarchies.

Invariant 2 *Let $C_1, \ldots, C_m$ be the root classes of the ISA relationship then $Ext_i \cap Ext_j = \emptyset$ for each i, j with $i \neq j$, $1 \leq i, j \leq m$.*

5.4 Schema and Database

We are now able to integrate the notions of values, types, objects, classes and inheritance we have dealt with in the previous sections. Chimera, as most object-oriented data models, distinguishes between the schema level (the time-invariant component) and the instance level (the time-varying one).

Definition 31 *(Schema). Given a set $\mathcal{AN}$ of attribute names, a set $\mathcal{TN}$ of type names, a set $\mathcal{MN}$ of method names, a set $\mathcal{CI}$ of class identifiers and a collection of basic domains $D_1, \ldots, D_n$, a Chimera schema is a tuple*

$$(VT, Cl, MCl, type, ISA)$$

where

- $VT \subseteq \mathcal{VT}$[30] *is a (finite) set of value types;*
- Cl *is a finite set of definitional components of classes;*
- $MCl \subseteq \mathcal{MC}$ *is a finite set of metaclasses;*
- $type : \mathcal{TN} \to \mathcal{VT}$ *is a total function on* $\mathcal{TN}$*;*
- $ISA : Cl \to 2^{Cl}$ *is a total function on Cl for which the following holds:*
  - *a) ISA is a DAG;*
  - *b) ISA is int-well-defined.*

All class and metaclass names are distinct and for each class the corresponding metaclass must exist. □

Definition 32 *(Database). Let S be a Chimera schema. A Chimera database over S is a tuple*

$$(OBJ, \pi, cval)$$

where

[30] $\mathcal{VT}$ is a denumerable infinite set of value types, implicitly defined starting from primitive domains, class names and attribute names according to Definition 3.

- *OBJ is a consistent set of objects;*
- *π is a pair of functions*

  $\pi_V : \mathcal{ET} \rightarrow 2^{\mathcal{V}}$, *value assignment*[31],
  $\pi_O : \mathcal{CI} \rightarrow 2^{\mathcal{OI}}$, *oid assignment*

  which handle class extents;
- *cval is a total function cval* $: Cl \rightarrow \mathcal{V}$ *such that* $\forall C \in Cl \quad cval(C)$ *is a legal value for* $stype(C.mc)$*, that is, the function cval assigns values to class attributes of* C*;*

such that:

(i) $\forall o = (i, v, c) \in OBJ$*, c must belong to* $Cl.id$*;*

(ii) $\forall C \in Cl$*, such that* $c = C.id$
$\{value(i) \mid i \in \pi(c)\} \subseteq [\![\, stype(c) \,]\!]_\pi \cup \bigcup_{c' \ s.t. \ c \in ISA(c')} [\![\, stype(c') \,]\!]_\pi$;

(iii) $\forall ET \in VT$*, extended value type,*
$\pi(ET) \subseteq [\![\, struct(ET) \,]\!]_\pi$;

(iv) $\forall C \in Cl$, $c = C.id$,
$cval(C).extent = \pi(c)$ *and*
$cval(C).proper\text{-}extent = \pi(c) \setminus \bigcup_{c' \ s.t. \ c \in ISA(c')} \pi(c')$;

(v) $\forall o \in OBJ$*, if* $o = (i, v, c)$*, and C is the class in Cl such that* $C.id = c$
$i \in cval(C).proper\text{-}extent$;

(vi) $\forall C \in Cl$, $\forall i \in cval(C).proper\text{-}extent$*, if* $o = (i, v, c)$*, then*
$C.id = c$;

(vii) *Invariant 1 must hold;*

(viii) *ISA is ext-well-defined;*

(ix) *Invariant 2 must hold.* □

6 Declarative Expressions

In this section we introduce Chimera rules, which are a mean to express declarative conditions on a database. They are used to express constraints and to specify the implementation of different class features. We consider a set of variables Var_T for each type $T \in \mathcal{T}$ of the language, to which the special variables $Self$ and $Class$ belong, used for denoting, respectively the object/the class on which the rule is being executed.

Definition 33 *(Terms). The set of Chimera terms Term is inductively defined as follows:*

- *each variable* $x \in Var_T$ *is a (atomic) term;*

[31] $\mathcal{V}$ is the set of values defined starting from basic values and object identifiers, according to Definition 8.

- *each basic value (constant) in* $ED_1, \ldots, ED_n$, *and the null value are (atomic) terms;*
- *let* $t_1, \ldots, t_n$, $n \geq 0$ *be terms, then* $\{t_1, \ldots, t_n\}$ *is a (complex) term;*
- *let* $t_1, \ldots, t_n$, $n \geq 0$ *be terms, then* $[t_1, \ldots, t_n]$ *is a (complex) term;*
- *let* $t_1, \ldots, t_n$, $n \geq 0$ *be terms, and* $a_1, \ldots, a_n \in \mathcal{AN}$ *be distinct labels, then* $(a_1 : t_1, \ldots, a_n : t_n)$ *is a (complex) term;*
- *let* t *be a term,* $a \in \mathcal{AN}$ *a label, then* $t.a$ *is a (complex) term;*
- *let* t *be a term,* n *a natural number, then* $t.n$ *is a (complex) term;*
- *let* $c \in \mathcal{CI}$ *be a class name,* $a \in \mathcal{AN}$ *a label, then* $c.a$ *is a (complex) term.* □

In addition to terms introduced by the definition above, Chimera supports a set of quite standard predefined operators that can be used to build terms. These predefined operators include arithmetic operators, set operators, list operators and aggregates on sets/lists.

Example 23 *The following are Chimera terms:*

- X;
- { `'bob'`, `'john'`, `'sue'` };
- $[true, X]$;
- $(a : X, b : true, c : 1627)$;
- $X.name$;
- $[true, X].1$;
- `person.averageAge`

where $X \in Var$, `person` $\in \mathcal{CI}$, $a, b, c, name, averageAge \in \mathcal{AN}$. ◇

According to the above definition, values (except oids) are terms, path expressions (built making use of the dot notation) are terms. In addition we consider a number of terms obtained using classical predefined operators for integers, reals, lists, sets. Of course there are type constraints for the applicability of such operators; we deal with typing of terms, making explicit the type of terms to which operators may be applied, in Section 6.1.1. Obviously, not all the terms obtained applying the operators on any two terms are correct terms.

We remark that oids, though values, are not included in terms. This is due to the fact that we do not allow oids to be manipulated explicitly by the user. Indeed in Chimera oids are a sort of system feature, not accessible by the user. In a user expression (such as a Chimera formula) the user may bind a variable to an oid, and then "access" the object denoted by this variable. We have indeed included in terms a typed set of variables, containing also variables of object types, that is variables denoting objects.

Method invocation (both at instance and at class level) are not included in terms because typically methods have side-effect and so they are not admitted in formulas (which are declarative expressions). Finally, we do not consider here complex aggregate operators, which in Chimera may be used to build terms.

Like terms, formulas are either atomic or complex. Atomic formulas are composed of a predicate symbol and a list of parameter terms. Predicate symbols are in a set Π. Π consists of a number of special predefined relations including comparison predicates, such as $<$ or $==$, and membership predicates (in). Class and type names, seen as unary predicate symbols, are included in Π. Finally, Π contains constraint names.

Definition 34 *(Atomic Formulas). Chimera atomic formulas are defined as follows:*

- Comparison Formulas

  if t_1, t_2 *are terms and* $op \in \{<, >, \geq, \leq, =, ==, ==_d\}$ *is a predefined predicate, then* t_1 *op* t_2 *is a comparison atomic formula;*

- Membership Formulas

  if t_1, t_2 *are terms, then* t_1 *in* t_2 *is a membership atomic formula;*

  if t *is a term and* $c \in \mathcal{CI}$ *is a class name, or* $c \in \mathcal{ET}$ *is a extended value type name, then* t *in* c *is a membership atomic formula;*

- Class Formulas

  if t *is a term and* c *is a class (or type) name, then* $c(t)$ *is a class formula*[32]*;*

- Constraint Formulas

  if $t_1, \ldots, t_n$ *are terms and con is a constraint symbol in* Π*, then* $con(t_1 \ldots, t_n)$ *is a constraint formula.* □

Example 24 *The following are Chimera formulas*

- *comparison formulas*

  $X = Y$
  $X == Z$[33]
  $Self.age > 18$
  `X.name = 'john'`

- *membership formulas*

  X *in* $Self.dependents$
  $\{1, 2\}$ *in* $X.dates$
  $Y.manager$ *in* `employee`

- *class formulas*

  `person(X)`
  `date((day:8, month:10, year:1969))`

- *constraint formulas*

  `tooLowIncome(X)`
  `invalidLifeExpentancy(Y).` ◇

[32] We will see in detail later how these formulas are used.
[33] In this formula we test for shallow value equality, while in the previous one we test for identity.

Complex formulas (or simply formulas) are obtained from atomic formulas and negated atomic formulas by means of conjunctions. All variables are assumed to be implicitly quantified as in Datalog.

Definition 35 *(Formulas). Formulas are inductively defined as follows:*

- *all atomic formulas are formulas;*
- *if F is an atomic comparison, membership or constraint formula*[34]*, then $\neg F$ is a (complex) formula;*
- *if F_1 and F_2 are formulas, then $F_1 \wedge F_2$ is a (complex) formula.* □

Example 25 *Examples of Chimera formulas are the following:*

- $\texttt{person(X)} \wedge \texttt{X.profession} =' \texttt{engineer}'$;
- $\texttt{person(X)} \wedge \neg\texttt{X in student} \wedge \texttt{X.income} < 6000$. ◇

We require that each formula contains exactly one class formula for each variable. This requirement is motivated in Section 6.1.1. In addition, we require formulas to be range restricted [22], to avoid formulas that are satisfied by an infinite set of instances.

Chimera rules are means for defining constraints, derived attributes (and views). By means of rules we may define the extent of a subclass by filtering out certain objects from the superclass and may define the value of an attribute intensionally, that is, without enumerating the value for every instance separately.

Definition 36 *(Rules). A Chimera rule is an expression of the form*

$$\textit{Head} \leftarrow \textit{Body}$$

where

- *Head is an atomic formula;*
- *Body is an arbitrary formula;*
- *each variable in the head occurs in the body.* □

Example 26 *Examples of Chimera rules are the following:*

- `engineer(X)` ← `employee(X), X.profession = 'engineer'`
  defining the population of a derived subclass;
- `Self.salary = 20000` ← `engineer(Self), Self.age` < `35`
  defining the implementation of a derived attribute;
- `Y in Self.children` ← `person(Self), person(Y), Y.father=Self`
  defining the implementation of a derived set attribute;

[34] Class formulas cannot be negated.

- `improperDate(Date)` $\leftarrow$ `Date.day` $<$ `0`

  defining a constraint;

- `tooLowIncome(N)` $\leftarrow$ `Self.income`$<$`50000, N = Self.name`

  defining a constraint. $\diamond$

Chimera rules must satisfy certain stratification and safety conditions [22].

6.1 Typing Issues

In this subsection we deal with the typing of declarative expressions. We first present rules for typing terms and formulas, making explicit type requirements we impose on such constructs. Subtyping is based on the well-known ideas of Cardelli object-oriented type system [6, 16, 17, 18], exploited in [4] for defining a type system for a deductive query language for the TM data model. This kind of typing is performed assuming that for each variable a type is known, thus we explicitly investigate this issue. Finally, we examine the class implementation and establish some criteria for the consistency of an implementation with respect to a signature.

6.1.1 Typing of Terms and Formulas

Table 3 presents the term typing rules. Typing rules for terms built with predefined operators are not presented here due to space limitations. They can be found in [35]. The following result holds.

Theorem 4 *For each term t in $Term$, $t \neq null$, the typing rules in Table 3 determine a unique type T for t.*

Table 4 specifies some conditions that must be satisfied for the correct application of predicates in Π to terms. Therefore, the following conditions exactly specify Chimera legal formulas. As far as class formulas are concerned we do not specify any type constraint for the application of class/type predicates because class formulas are the mean to specify the type of a variable. We only require that class formulas are applied to variables.

The above typing rules are based on the existence of a basis, that is, of a set of statements of the form $X \in Var_T$, where X is a variable and T is a type. Other typing rules make use of this basis. In the following we examine how these bases are obtained.

When writing a Chimera formula, we must explicitly state the type of each variable used in the expression. This is accomplished in Chimera by using class formulas. Indeed, in Chimera we use class formulas as a typing mechanism. A class formula has the form $c(X)$ where $c \in \mathcal{CI}$ is a class name and X is a variable. The meaning of this formula is to assign the type corresponding to c to variable X and to state that the objects, to which X may be instantiated, must be members of the class identified by c. This formula may be seen as a shorthand for the expression $X : c \wedge X \texttt{ in } \pi(c)$ where $X : c$ is a type declaration, and `in` is the membership predicate. Class formulas are built from class or type names representing unary predicate symbols. Thus, to state that a variable X is of type T (that is $X \in Var_T$) we add to our expression the class formula $T(X)$. Note that this has also the effect to state that $X \in [\![T]\!]$, thus providing a domain for the evaluation of the expression.

For each type declaration $c(X)$ we add $X \in Var_c$ to the basis, and, to properly type the pseudovariable $Self$, if a rule appears in the context of a class c we add $Self \in Var_c$ to the basis.

$$\frac{}{null : T} \quad T \in \mathcal{T}$$

$$\frac{X \in Var_T}{X : T}$$

$$\frac{t \in dom(D_i)}{t : D_i} \quad D_i \in \mathcal{BVT}$$

$$\frac{t_i : T_i \quad (1 \leq i \leq n)}{\{t_1, \ldots, t_n\} : \textit{set-of}(T)} \quad T = \bigsqcup_{i=1}^{n} T_i$$

$$\frac{v_i : T_i \quad (1 \leq i \leq n)}{[t_1, \ldots, t_n] : \textit{list-of}(T)} \quad T = \bigsqcup_{i=1}^{n} T_i$$

$$\frac{t_i : T_i \quad (1 \leq i \leq n)}{(a_1 : t_1, \ldots, a_n : t_n) : \textit{record-of}(a_1 : T_1, \ldots, a_n : T_n)} \quad a_1, \ldots, a_n \in \mathcal{AN}$$

$$\frac{t : \textit{record-of}(a_1 : T_1, \ldots, a_n : T_n)}{t.a_i : T_i} \quad i = 1 \ldots n$$

$$\frac{t : \textit{list-of}(T)}{t.n : T} \quad n \in \mathbb{N}$$

$$\frac{t : Tn, Tn \in \mathcal{TN}, type(Tn) = \textit{record-of}(a_1 : T_1, \ldots, a_n : T_n)}{t.a_i : T_i} \quad i = 1 \ldots n$$

$$\frac{t : Tn, Tn \in \mathcal{TN}, type(Tn) = \textit{list-of}(T)}{t.n : T} \quad n \in \mathbb{N}$$

$$\frac{t : c, \quad c \in \mathcal{CI}, \quad stype(c) = \textit{record-of}(a_1 : T_1, \ldots, a_n : T_n)}{t.a_i : T_i} \quad i = 1 \ldots n$$

$$\frac{c \in \mathcal{CI},\ C \in \mathcal{C},\ C.id = c,\ C.mc = c',\ stype(c') = \textit{record-of}(a_1 : T_1, \ldots, a_n : T_n)}{c.a_i : T_i} \quad i = 1 \ldots n$$

Table 3: **Chimera term typing rules**

The scope of a type assignment for a variable is that of an expression (a formula). Each Chimera formula must contain a class formula for each variable. Thus, each variable is associated with a unique type, with respect to which type checking is performed. Declaring a variable X of type T influences type checking in that the only features available for X are those of type T. That is, type checking of the expression, in which X appears, is done regarding X as a term of type T. We say that T is the *static type* of variable X, because it is the type with respect to which the static type checking of the expression containing X is done.

At execution time variable X is instantiated with a value of type T. Note that when T is an object type, the variable is instantiated with the identifier of an object member of the class identified by T. This means that the variable may also be instantiated with the identifier of an object instance of a subclass T_s of that class. In this case, we say that T_s is the *dynamic type* of variable X. The dynamic type is the type used for choosing the appropriate attribute/method definition to be used on the object denoted by X. *Late binding* indeed requires that on X we use the most specific implementation for the object on which X is instantiated, that is, the implementation of T_s. Note that the chosen implementation is always the implementation provided by the most specific class of the object, independently from the type declared for variable X.

$$\frac{t_1, t_2 : T}{t_1\ op\ t_2} \quad T \in \{\texttt{integer}, \texttt{real}, \texttt{character}, \texttt{string}\},\ op \in \{<, >, \geq, \leq\}$$

$$\frac{t_1 : T_1,\ t_2 : T_2}{t_1 = t_2} \quad T_1 \sqcup T_2 \text{ is defined}$$

$$\frac{t_1 : T_1,\ t_2 : T_2}{t_1 == t_2} \quad T_1, T_2 \in \mathcal{OT},\ T_1 \sqcup T_2 \text{ is defined}$$

$$\frac{t_1 : T_1,\ t_2 : T_2}{t_1 ==_d t_2} \quad T_1, T_2 \in \mathcal{OT},\ T_1 \sqcup T_2 \text{ is defined}$$

$$\frac{t_1 : T_1, t_2 : \textit{set-of}(T_2)}{t_1\ in\ t_2} \quad T_1 \sqcup T_2 \text{ is defined}$$

$$\frac{t_1 : T_1, t_2 : \textit{list-of}(T_2)}{t_1\ in\ t_2} \quad T_1 \sqcup T_2 \text{ is defined}$$

$$\frac{t_1 : \textit{set-of}(T_1), t_2 : \textit{set-of}(T_2)}{t_1\ in\ t_2} \quad T_1 \sqcup T_2 \text{ is defined}$$

$$\frac{t_1 : \textit{list-of}(T_1), t_2 : \textit{list-of}(T_2)}{t_1\ in\ t_2} \quad T_1 \sqcup T_2 \text{ is defined}$$

$$\frac{t : T, \quad c \in \mathcal{CI}}{t\ in\ c} \quad T \in \mathcal{OT}, T \sqcup c \text{ is defined}$$

$$\frac{t : T, \quad ET \in \mathcal{ET}}{t\ in\ ET} \quad T \in \mathcal{VT}, T \sqcup ET \text{ is defined}$$

$$\frac{X \in Var, \quad T \in \mathcal{T}}{T(X)} \quad Var = \bigcup_T Var_T$$

Table 4: **Chimera formula typing rules**

6.1.2 Rule Typing

We now examine with a greater detail the rules that constitute a class implementation. As far as population implementation is concerned, it consists of a set of rules whose heads are class formulas on the class name whose population is being defined. For example

`employee(X)` $\leftarrow$ `person(X), X.profession='employee'`

is a rule defining the population of the class `employee`, subclass of `person`.

An attribute implementation is a set of rules of one of the following forms

$$Self.a = term \leftarrow body$$
$$term\ in\ Self.a \leftarrow body$$

where $Self$ is the pseudovariable used for denoting the object on which the rule is being executed.

A constraint implementation is a set of rules of the form

$$constraint\_formula \leftarrow body$$

where $constraint\_formula$ is a constraint name applied to a list of parameters, that is $con(t_1, \ldots, t_n)$.

An operation implementation is an expression of the form

$$\textit{op-name} : condition \rightarrow \textit{op-code}$$

where *op-name* is the operation name in $\mathcal{MN}$ applied to a list of parameters, *condition* is a Chimera formula, and *op-code* is a sequence of update operations. We do not examine in detail the syntax of update operations, nor we deal with typing issues related to those constructs, because it is beyond the scope of this paper. We only require that for operation input parameters each parameter is used in *condition* according to the type declared for it[35], while for operation output parameters the type of the term returned as output may be a subtype of the type declared as return type in the operation signature.

Definition 37 *A class implementation is a set of rules, each having one of the following forms*

1. $class\_formula \leftarrow body$

   expressing the population of the class;

2. $equality\_formula \leftarrow body$

   expressing the implementation of a derived attribute;

3. $membership\_formula \leftarrow body$

   expressing the implementation of a derived set valued attribute;

4. $constraint\_formula \leftarrow body$

   expressing a constraint

plus a set of operation implementations, namely expressions of the form

5. *op-name* : *condition* $\rightarrow$ *op-code*. □

Example 27 *The following is an example of class implementation.*

```
Self.age = X ← X = 1994 - Self.birthday.year
tooLowIncome(N) ← Self.income < 5000, N = Self.name
changeIncome(Amount): integer(New), New = Self.income + Amount
                        → modify(person.income,Self,New)
```
◇

As already stated, a number of conditions must be verified by a set of rules of this form to be a correct implementation for a class C. These conditions are as follows. First of all, the implementation must contain at least a rule for each class feature. Moreover,

1. the $class\_formula$ must denote the class whose implementation is being defined;
2. $equality\_formula$ must be of the form

   $$Self.a = term$$

   with a derived attribute for the class whose implementation is being defined;
3. $membership\_formula$ must be of the form

   $$term\ in\ Self.a$$

[35] Note that the formula that constitutes the condition in an operation implementation may not have class formulas for variables identifying input parameters of the operations, since these variables are already typed by the operation signature.

with a derived attribute for the class whose implementation is being defined;

4. *constraint_formula* must be constructed with a constraint of the class, applied to a number of parameters corresponding to that declared in the signature.

For operation implementation (5.) we require that *op-name* corresponds to an operation of the class, applied to a number of parameters corresponding to those declared in the signature.

Conditions 1—4 above, however, are not sufficient to guarantee that the set of rules provides a correct implementation for a class because they do not take into account type compatibility. Thus we have the following additional conditions related to typing of rules:

1. if the head of the rule is $c(X)$, and the type declared for X in rule body is T, then $c \leq_T T$ must hold[36];

2. if the head of the rule is $Self.a = term$ and the type deduced for $term$, starting from the type declaration in rule body, is T, then $T \leq_T T^a$ must hold, where T^a is the domain specified for attribute a in the class signature;

3. if the head of the rule is $term\ in\ Self.a$ and the type deduced for $term$, starting from the type declaration in rule body, is T, then $set\text{-}of(T) \leq_T T^a$ must hold, where T^a is the domain specified for attribute a in the class signature;

4. if the head of the rule is $con(t_1, \ldots, t_n)$ and the type deduced for each t_i, starting from the type declaration in rule body, is T_i, $i = 1, \ldots, n$, then $T_i \leq_T T_i^c$ must hold, where the signature for constraint con in the class signature is $c \rightarrow T_1^c \times \ldots \times T_n^c$;

5. if the operation signature is $T_1^o \times \ldots \times T_n^o \rightarrow T^o$, then type checking of *condition* is done taking $T_1^o, \ldots, T_n^o$ as types for input parameters (for which class formulas may be missing), while the type T deduced for the output parameter, starting from the type declaration in rule body, is a subtype of T^o, that is $T \leq_T T^o$ must holds.

Example 28 *Consider the class signature of Example 10 and the class implementation of Example 27. Since pop = ext, no population implementation is provided. The only derived attribute in struct is (*`age, integer`*, der), whose implementation is specified by the rule*

`Self.age = X ← X = 1994 - Self.birthday.year`.

Since $Self$ denotes the object on which the rule is executed, it is implicitly typed to the class to which the rule belongs, in that case class `person`*. Thus, according to* `person` *signature,* `Self.birthday` *has type* `date`*, and, since type(*`date`*)=*`record-of(day:integer, month:integer, year:integer)`*, the term* `Self.birthday.year` *has type* `integer`*. Since also the value* `1994` *has type* `integer`*, the term* `1994 - Self.birthday.year` *has type* `integer` *too. Thus, variable* `X` *has then type* `integer`*,which is exactly the type declared for attribute* `age`*.*

As far as operation implementation is concerned, operation `changeIncome: integer → person` *is implemented by the expression*

```
changeIncome(Amount): integer(New), New=Self.income + Amount →
                      modify(person.income,Self,New)
```

where the implicit output parameter is $Self$, of type `person`*, while the condition is well-typed for input parameter* `Amount` *of type* `integer`*.*

[36] We may insert in a derived class only elements extracted from one of its superclasses.

Finally, consider the constraint `tooLowIncome:person` $\rightarrow$ `string`*, implemented by the rule*

`tooLowIncome(N)` $\leftarrow$ `Self.income <5000, N=Self.name`.

Since $Self$ *has type* `person`, `Self.income` *has type* `integer`*, while* `Self.name` *has type* `string`*. Thus, the formula in the rule body is well typed and* `N` *has type* `string`*, coherently with the constraint signature.* $\diamond$

7 Conclusions

In this paper we have presented a formal definition of the the Chimera data model. Table 5 provides a comparison among formal models for object-oriented databases so far proposed. In particular, we have compared the formal models under the following aspects: whether complex objects are supported; whether the model supports single or multiple inheritance; whether the model supports both the notions of type and class[37]; whether relations are included in the model (thus coupling an object model with a value-oriented one); whether typing issues are covered. Most models originate from the work of Beeri [7], which has been the first to clarify several issues on object-oriented data models. Furthermore, some proposals formalize a core object-oriented data model as a basis for addressing other issues, such as particular forms of integrity constraints in [46], a behavior specification in [38], an algebra and communication issues in [42]. Also the model developed in [51] may be seen as a basis for the development of an algebra [50]. Thus, those formal models do not adequately cover all features of a real object-oriented data model.

Other formal models for object-oriented databases have been developed that we have not considered in the comparison. However, most of them are concerned with a specific aspect and only consider a partial view. In [3] type hierarchies and subtyping problems are considered, but other aspects are not covered. In [49] a simple object model is formally specified as a basis for addressing query issues. [33] and [45] are mainly concerned with the specification of object behavior.

	[1, 41]	[46]	[38]	[51]	[42]	Our model
Complex Objects	Partially$^{(1)}$	YES	Partially$^{(1)}$	YES	YES	YES
Inheritance	Single	Multiple	No	Single	Multiple	Multiple
Classes & Types	Types only	Both	Sorts only	Both	Both	Both
Relations	NO	NO	YES	NO	NO	NO
Schema - Instance Levels	YES	YES	YES	YES	YES	YES
Feature Refinement	YES	NO	NO	Attributes only	YES	YES
Type System	NO	NO	NO	NO	NO	YES
Class Features	NO	NO	NO	NO	NO	YES
Named Types	NO	NO	NO	NO	NO	YES

Legenda: $^{(1)}$ Partially denotes the non-complete orthogonality in applying constructors to arbitrary values

Table 5: **Comparison with other formal models for OO databases**

This comparison points out that our model covers a number of modeling features not covered by other models, since we consider a very rich and complex object-oriented data models, and

[37] We remark that in models supporting both the notion of type and that of class, the two notions seldom have the same meaning in different models.

we have developed a type system for the language. We would like to point out, moreover, that, though the Chimera model is a quite complex model, most of the features it supports are rather similar to features supported by modern object-relational DBMSs, like, for instance, DB2 [24] and Illustra [37]. In particular, those data models support: user-defined value types (primitive type extension in ORDBMSs); classes; triggers and integrity constraints. In addition, Chimera supports deductive rules, which allow to express constrained types, derived attributes, and views. However, these features are orthogonal, and thus non-redundant, with respect to the features listed above, supported by ORDBMSs.

The work may be extended along several ways. A formal definition of a view mechanism for Chimera is presented in [36] while a temporal extension of the object-oriented data model presented in this paper has been developed [9]. We are currently investigating the problem of trigger and constraint redefinition in the context of object-oriented data models [11]. From the typing viewpoint, we are going to extend our work considering typing of queries and transactions (including also method invocations) and typing of programs (API). We may also consider a kind of type inference, allowing variables in Chimera formulas to be indirectly typed using information in their signature definition.

Acknowledgments

We wish to thank Stefano Ceri, Letizia Tanca and all the IDEA group at Politecnico di Milano for helpful discussions on the subject of this paper. Claudio Bettini provides us with some insights on knowledge representation systems.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Chapter 21: Object Databases. Addison-Wesley, 1995.

[2] R. Agrawal and N. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *Proc. ACM-SIGMOD Int'l Conf. on Management of Data*, Portland, Oregon, 1989, pages 36-45.

[3] A. Albano, L. Cardelli, and R. Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.

[4] R. Bal and H. Balsters. A Deductive and Typed Object-Oriented Language. In S. Tsur, S. Ceri, and K. Tanaka, editors, *Proc. Third Int'l Conf. on Deductive and Object-Oriented Databases*, Lecture Notes in Computer Science 760, pages 340–359, 1993.

[5] H. Balsters, R. A. de By, and R. Zicari. Typed Sets as a Basis for Object-Oriented Database Schemas. In O. Nierstrasz, editor, *Proc. Seventh European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science 707, 1993.

[6] H. Balsters and M. Fokkinga. Subtyping can have a Simple Semantics. *Theoretical Computer Science*, 87:81–96, September 1991.

[7] C. Beeri. Formal Models for Object Oriented Databases. In W. Kim et al., editor, *Proc. First Int'l Conf. on Deductive and Object-Oriented Databases*, pages 370–395, 1989.

[8] E. Bertino, C. Cascella, and G.Guerrini. Composite Object Handling Through Triggers. Technical Report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 1997.

[9] E. Bertino, E. Ferrari, G. Guerrini. T_Chimera: A Temporal Object-Oriented Data Model. To Appear in *Theory and Practice of Object Systems*, John Wiley & Sons. 1997.

[10] E. Bertino and G. Guerrini. Objects with Multiple Most Specific Classes. In W. Olthoff, editor, *Proc. Ninth European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science 952, pages 102–126, 1995.

[11] E. Bertino, G. Guerrini, and I. Merlo. Trigger Inheritance and Overriding in an Active Object Database System. Technical Report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova. Submitted for publication, 1997.

[12] E. Bertino, G. Guerrini, and D. Montesi. Deductive Object Databases. In M. Tokoro and R. Pareschi, editors, *Proc. Eighth European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science 821, pages 213–235, 1994.

[13] E. Bertino and L.D. Martino. *Object-Oriented Database Systems - Concepts and Architecture.* Addison-Wesley, 1993.

[14] R. Breitl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams. The GemStone Data Management System. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 283–308. Addison-Wesley, 1989.

[15] S.J. Cannan and G.A.M. Otten. *SQL - The Standard Handbook.* McGraw-Hill, 1992.

[16] L. Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*,76:138–164, 1988.

[17] L. Cardelli. Types for Data Oriented Languages. In J. W. Schmidt, S. Ceri, and M. Missikoff, editors, *Proc. First Int'l Conf. on Extending Database Technology*, Lecture Notes in Computer Science 303, pages 1–15, 1988.

[18] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polimorphism. *Computing Surveys*, 17:471–522, 1985.

[19] F. Cattaneo. Type-safe Covariant Redefinition in the Chimera Type System. Technical Report, Politecnico di Milano, October 1995.

[20] R. Cattel. *The Object Database Standard: ODMG-93.* Morgan-Kaufmann, 1996.

[21] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990.

[22] S. Ceri and R. Manthey. Consolidated Specification of Chimera. Technical Report, IDEA.DE.2P.006.01, November 1993.

[23] S. Ceri and R. Manthey. Chimera: A Model and Language for active DOOD Systems. In *Extending Information System Techology - Second International East-West Database Workshop*, Lecture Notes in Computer Science, pages 9–21, 1994.

[24] D. Chamberlin. *Using the New DB2 - IBM's Object-Relational Database System.* Morgan-Kaufmann, 1996.

[25] A. Coen Porisini, L. Lavazza, and R. Zicari. Static Type Checking of Object-Oriented Databases. Technical Report 91-060, Politecnico di Milano, October 1991.

[26] R. Connor and R. Morrison. Subtyping without Tears. In *Proc. Fifteenth Australian Computer Science Conference*, pages 209–225, 1992.

[27] P. Dechamboux, M. Lopez, and C. Roncancio. The Data Model of the Peplom DBPL. Technical report, IDEA.WP.1B.001, October 1992.

[28] O. Deux et al. The Story of 0_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.

[29] D. H. Fishman et al. Overview of the Iris DBMS. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 219–250. Addison-Wesley, 1989.

[30] L. J. Gallagher. Object SQL: Language Extensions for Object Data Management. In *Proc. First International Conference on Information and Knowledge Management (CIKM)*, Baltimore, Maryland, November 1992.

[31] N. Gehani and H. Jagadish. Ode as an Active Database: Constraints and Triggers. In *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 327–336, 1991.

[32] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[33] G. Gottlob, G. Kappel, and M. Schrefl. Semantics of Object-Oriented Data Models - The Evolving Algebra Approach. In J. W. Schmidt and A. Stogny, editors, *Next Generation System Technology*, Lecture Notes in Computer Science 504, pages 144–160, 1991.

[34] G. Guerrini. An Active and Deductive Object-Oriented Data Model. PhD Thesis in Preparation, Università di Genova, 1997.

[35] G. Guerrini, E. Bertino, and R. Bal. A Formal Definition of the Chimera Object-Oriented Data Model. Extended version of this paper, 1995.

[36] G. Guerrini, E. Bertino, B. Catania, J. Garcia-Molina. A Formal Model of Views for Object-Oriented Database Systems. To Appear in *Theory and Practice of Object Systems*, John Wiley & Sons. 1997.

[37] *Illustra User's Guide*, release 2.1. Oakland, CA: Illustra Information Technologies.

[38] A. Kemper and G. Moerkotte. A Formal model for Object-Oriented Databases: The First Step. Technical report, University of Karlsruhe, 1991.

[39] W. Kim, E. Bertino, and J. Garza. Composite Objects Revisited. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, 1989.

[40] W. Kim et al. Features of the ORION Object-Oriented Database System. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 251–282. Addison-Wesley, 1989.

[41] C. Lecluse, P. Richard, and F. Velez. 0_2, an Object-Oriented Data Model. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 257–276. Addison-Wesley, 1990.

[42] L. Liu. *A Formal Approach to Structure, Algebra and Communication of Complex Objects.* PhD thesis, Katholieke Universiteit Brabant Tilburg, November 1992.

[43] B. Meyer. *Eiffel: The Language.* Prenctice Hall Internal - Object-Oriented Series, Second Edition, 1992.

[44] C. Peltason, A. Schmiedel, C. Kindermann, and J. Quantz. The BACK System Revisited. Technical Report KIT - Report 75, Technische Universitat Berlin, 1989.

[45] R. J. Peters. *TIGUKAT: A Uniform Behavioral Objectbase Managent System.* PhD thesis, Department of Computing Science, University of Alberta, April 1994.

[46] K. D. Schewe and B. Thalheim. Fundamental Concepts of Object Oriented Databases. *Acta Cybernetica*, 11(1-2):49–83, 1993.

[47] M. Scholl and H. Schek. A Relational Object Model. In S. Abiteboul and P.C. Kanellakis, editors, *Proc. Third International Conference on Database Theory*, pages 89–105, 1990.

[48] M. Stonebraker, L. Rowe, and M. Hirohama. The Implementation of Postgres. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, 1990.

[49] D. Straube and M. Ozsu. Queries and Query Processing in Object-Oriented Database Systems. *ACM Transactions on Information Systems*, 8(4):387–430, October 1990.

[50] G. Vossen and K. U. Witt. FASTFOOD: A Formal Algebra over Sets and Tuples for the FOOD Object-Oriented Data Model. Technical report, Arbeitsgruppe Informatik, Justus Liebig Universitat, Giessen, August 1991. TR n. 9103.

[51] G. Vossen and K. U. Witt. Objectbase Schemata and Objectbases in the FOOD Model. Technical report, Arbeitsgruppe Informatik, Justus Liebig Universitat, Giessen, June 1991. TR n. 9101.