A Formal Temporal Object-Oriented Data Model

Elisa Bertino[1] Elena Ferrari [1] Giovanna Guerrini [2]

[1] Dipartimento di Scienze dell'Informazione - Università di Milano
Via Comelico, 39/41 - I20133 Milano, Italy
{bertino@hermes.mc.dsi.unimi.it, ferrarie@dsi.unimi.it}

[2] Dipartimento di Informatica e Scienze dell'Informazione - Università di Genova
Viale Benedetto XV, 3- I16132 Genova, Italy
guerrini@disi.unige.it

Abstract

Temporal databases are an active and fast growing research area. Although many extensions to the relational data model have been proposed in order to incorporate time, there is no comparable amount of work in the context of object-oriented data models. This paper presents T_Chimera, a temporal extension of the Chimera object-oriented data model. The main contribution of this work is to define a formal temporal object-oriented data model and to address on a formal basis several issues deriving from the introduction of time in an object-oriented context.

1 Introduction

Time is an important aspect of most real-world phenomena. Conventional database systems do not offer the possibility of dealing with time-varying data. The content of a database represents a *snapshot* of the reality in that only the current data are recorded, without the possibility of maintaining the complete history of data over time. If such a need arises, data histories must be managed at application program level, thus, making the management of data very difficult, if at all possible. To overcome the lack of effective time support in database systems, in the past years there has been a growing interest in extending data models to deal with temporal aspects [18, 20]. Most proposals are temporal extensions of the relational data model [8, 9, 16]. In particular, the temporal data models proposed in the literature extend the relational model by adding a temporal component associated with tuples or attribute values. The extensions of the relational model to handle time can be classified in two main categories. The *tuple timestamping* approach uses normalized (1NF) relations in which special time attributes are added [16]. The *attribute timestamping* approach uses non-normalized (N1NF) relations and attaches time to attribute values [9]. One of the most common approaches [8, 9] views attribute values as partial functions from the time domain to the attribute value domain.

Research on temporal object-oriented databases is still in its early stage. Although various object-oriented temporal models have been proposed [17], there is no amount of theoretical work comparable to the work reported for the relational model. For example, Wuu and Dayal temporal extension of the OODAPLEX model [21] addresses several problems related to the introduction of time in an object-oriented context but not on a formal basis. By contrast, a great amount of theoretical research is being carried on in the area of non-temporal object-oriented programming languages and databases and theoretical foundations are being established [1, 3, 14]. In particular, issues concerning type systems and type checking have been widely investigated in a non-temporal object-oriented framework [2, 5]. These issues, however, have never been addressed in the framework of temporal object-oriented data models. Thus, the extension of established foundations to temporal object-oriented data models still needs deep investigations.

In this paper we present the T_Chimera data model, a temporal extension of the Chimera data model. Chimera is an object-oriented, deductive, active data model, being developed as part of the ESPRIT Project Idea P6333 [12]. It provides all concepts commonly ascribed to object-oriented data models, such as: object identity, complex objects, user-defined operations, classes and inheritance. Moreover, it provides capabilities for defining deductive rules, that can be used to define views and integrity constraints, to formulate queries, to specify methods to compute derived information. Finally, it supports a powerful language for defining triggers. A formal model has been defined for Chimera, providing a formal foundation for the various features of the Chimera data model [12].

The current paper focuses on selected features of T_Chimera. Its main contribution is to address on a formal basis several issues deriving from adding temporal capabilities to an object-oriented model. First, the notion of temporal type is introduced. Temporal types handle in a uniform way temporal and non-temporal domains. Then, we provide a formal definition of classes and objects. The notion of consistency of an object with respect to its class is specified, keeping into account that both the object state and the classes the object belongs to[1] vary over time. Finally, a formal treatment of other features of the model, like object equality, referential integrity and inheritance is presented. Among the novel aspects of the proposed model, let us cite its support for immutable, static and temporal attributes, the availability of different notions of object equality and its notion of substitutability through coercion from temporal to non-temporal domains.

This paper is organized as follows. In the remainder of this section we survey related works. Section 2 provides a brief overview of the Chimera data model. Section 3 formally introduces T_Chimera types and values, while Sections 4 and 5 deal with classes and objects respectively. Inheritance is considered in Section 6. Finally, Section 7 concludes the paper and outlines future work.

1.1 Related works

In this section we compare some of the temporal object-oriented data models proposed so far. Some of the approaches considered here are compared under a quite different perspective by Snodgrass [17]. In [17] the emphasis is on temporal object-oriented query languages, while we consider here only data model characteristics. Moreover, in [17] only the temporal characteristics are compared, disregarding the object-oriented ones, whereas we consider both. Tables 1 and 2 compare the existing proposals along a number of dimensions. These dimensions can be classified into three broad categories: *(i)* object-oriented characteristics modeled; *(ii)* temporal characteristics modeled (that is, notions of time supported); *(iii)* how these characteristics are integrated, that is, how time is associated with objects.

For the object-oriented characteristics, we consider in the table only two dimensions: whether values are distinguished from objects (and types from classes at the intensional level) [3, 14], and whether class features are supported. As a general remark, we point out that most proposed approaches focus on the temporal characteristics of the model and consider rather poor (simple) object models.

[1] Objects can migrate during their lifetime from one class to another.

	oo data model	time structure	time dimension	values & objects	class features
[21]	OODAPLEX	user-defined	arbitrary[1]	objects	NO[2]
[6]	OODAPLEX	linear	valid	objects	NO[2]
[11]	TIGUKAT	user-defined	valid	objects	NO
[13]	MAD	linear	valid	objects	NO
[19]	OSAM*	linear	valid	objects	NO
[15]	3DIS	linear	valid	objects	NO
[7]	generic	linear	valid	objects	NO
Our model	Chimera	linear	valid	both	YES

Legenda:
[1] One single time dimension is considered, but it can be interpreted either as transaction or as valid time.
[2] OODAPLEX supports metadata, but neither [21] nor [6] consider them.

Table 1: Comparison among the existing temporal object-oriented data models (I)

Concerning the temporal aspects, we identify the time domain considered by each approach. In particular, we consider time structure and time dimension [17]. Most models support a linear discrete time structure, whereas only few of them [21, 11] model a user-defined hierarchy of time types. Two time dimensions are of interest in temporal databases: *valid* time (the time a fact was true in reality) and *transaction* time (the time the fact was stored in the database). Most models consider only the first one. Although our model focuses on a single time structure and dimension, it can be easily extended to different notions of time.

Let us now consider how temporal and object-oriented characteristics are combined. As shown by the table, some approaches associate a timestamp with the whole object state, whereas others associate a timestamp separately with each object attribute. Among the approaches associating timestamps with single attribute values, the majority regards the value of a temporal attribute as a function from a temporal domain to the set of legal values for the attribute. Another important characteristic, along which the existing approaches can be classified, is whether temporal, immutable and non-temporal attributes are supported. A *temporal* (or historical) attribute is an attribute whose value may change over time, and whose values at different times are recorded in the database. An *immutable* attribute is an attribute whose value cannot be modified during the object lifetime, whereas a *non-temporal* (or static) attribute is an attribute whose value can change over time, but whose past values are not meaningful for the application at hand, and are thus not stored in the database. Immutable attributes can be regarded as a particular case of temporal ones, since their value is a constant function from a temporal domain. For a model like ours, that distinguishes between objects and values, we think that the notion of non-temporal attribute is much more relevant in practice than the notion of immutable one. Finally, we have examined whether the considered approaches model the history of associations between an object and its type (class). Indeed, an important dynamic aspect of object-oriented databases is that an object can dynamically change type, by specializing or generalizing its current one [22] (often this type change is referred to as object *migration*). Thus, we distinguish the approaches keeping track of the dynamic links between an object and its most specific class from those that does not consider this aspect.

As a final remark, let us mention that, as noticed in [17], in contrast to temporal relational data models, the specification of temporal object-oriented data models is in most cases informal. In particular, none of the considered proposals addresses the issues resulting from the introduction

	what is timestamped	temporal attribute values	kinds of attributes	histories of object types
[21]	arbitrary	functions[1]	temporal + immutable	YES
[6]	attributes	functions[1]	temporal + immutable	NO
[11]	arbitrary	sets of pairs	temporal + immutable	YES
[13]	objects	atomic valued[2]	temporal + immutable	NO
[19]	objects	atomic valued[2]	temporal + immutable	NO[4]
[15]	attributes	sets of triples[3]	temporal	NO
[7]	attributes	functions[1]	temporal + immutable	YES
Our model	attributes	functions[1]	temporal + immutable + non-temporal	YES

Legenda:
[1] With the term *functions* we have denoted functions from a temporal domain.
[2] Time is associated with the entire object state.
[3] The triple elements are (oid, attribute name, attribute value); a time interval and a version number are associated with each element of the triple.
[4] The information is not associated to objects, it can however be derived from the histories of object instances.

Table 2: Comparison among the existing temporal object-oriented data models (II)

of time in an object-oriented data model on a formal basis. Moreover, only Wuu and Dayal in [21] discuss the consistency of an object with respect to the classes it belongs to, and only few proposals [6, 11, 21] consider the impact of inheritance in a temporal framework. However, none of them addresses the problems concerned with domain refinement and substitutability.

2 The Chimera data model

In this section we briefly review the main concepts of the Chimera data model that are relevant to this work. In Chimera, as in many other object-oriented data models, **objects** are abstraction of real-world entities (such as a person or an employee). Each object has a unique system defined object identifier (*oid*). The *oid* is assigned automatically by the system upon the object creation and remains immutable for the lifetime of the object. Properties of objects are described by means of *attributes*. Attribute values may change over time, without changing the object identity. Objects can be manipulated by means of *operations*, which can be built-in or user-defined. Objects having similar properties and behavior are grouped into **classes** organized in inheritance hierarchies. Chimera provides a uniform notion of object, in that classes are themselves objects. Then, classes are characterized by their set of attributes and operations, called c-attributes and c-operations respectively. C-attributes and c-operations are a mean for associating a value or an operation with an entire class rather than with its instances. C-attributes can be used to record statistical information, like the minimum salary or the average age of employees, while c-operations can be used to manipulate such values.

Both *primitives* and *complex* **values** are supported. Complex values are built by using constructors like set, list and record. Complex values are defined as instances of value types. Thus, value types provide the same function as concrete types commonly found in programming languages. Chimera supports both values and objects. The constructors provided by Chimera can be applied to atomic values, complex values and objects. Therefore, constructors can be nested and a complex value may refer to an object. The main differences between Chimera values and Chimera objects can be summarized 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 but their identity does not depend on the attribute values. Changing the values of an object 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 pre-defined operations, which are provided by Chimera.

3 Types and values

In this section we introduce the set of *T*_Chimera types and values. *T*_Chimera types extend the set of Chimera types [12], with a set of *temporal types*. Temporal types are introduced to type, in a uniform way, variables for which the history of changes over time is recorded and variables for which only the current value is kept.

3.1 T_Chimera types

In the following we briefly review the Chimera types, then we extend them to type historical variables.

In the remainder of the discussion we denote with $\mathcal{OI}$ a set of object identifiers, with $\mathcal{CI}$ a set of class identifiers, that is, class names. Moreover, we denote with $\mathcal{AN}$ a set of attribute names and with $\mathcal{MN}$ a set of method names. Table 3 summarizes the functions used in defining the model. For each of them the table reports the name, the signature, that is, the type of the input parameters and of the output parameter and a brief description. The meaning of each function will be clarified as soon as the functions will be introduced in the following sections.

In Chimera the existence of a finite set $\mathcal{BVT} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$ of basic predefined value types is postulated, containing at least the types `integer`, `real`, `bool`, `character` and `string`. Moreover, Chimera allows class names to be used in the definition of types. The following definition states that each class name is a (object) type.

Definition 3.1 (Object Types) *[12] The set of Chimera object types $\mathcal{OT}$ is defined as the set of class identifiers $\mathcal{CI}$.*

Chimera supports *structured types* such as sets, lists and records. A set type is denoted *set-of*(T), where T is a type. Instances of *set-of*(T) are sets of instances of type T. A list type

Name	Signature	Description
$\mathcal{T}^{-}$	$\mathcal{TT} \rightarrow \mathcal{CT}$	returns the static type corresponding to a temporal type
π	$\mathcal{CI} \times \mathcal{TIME} \rightarrow 2^{\mathcal{OI}}$	returns the extent of a class at a given instant
type	$\mathcal{CI} \rightarrow \mathcal{T}$	returns the structural type of a class
h_type	$\mathcal{CI} \rightarrow \mathcal{T}$	returns the historical type of a class
s_type	$\mathcal{CI} \rightarrow \mathcal{T}$	returns the static type of a class
h_state	$\mathcal{OI} \times \mathcal{TIME} \rightarrow \mathcal{V}$	returns the historical value of an object
s_state	$\mathcal{OI} \rightarrow \mathcal{V}$	returns the static value of an object
o_lifespan	$\mathcal{OI} \rightarrow \mathcal{TIME} \times \mathcal{TIME}$	returns the lifespan of an object
m_lifespan	$\mathcal{OI} \times \mathcal{CI} \rightarrow \mathcal{TIME} \times \mathcal{TIME}$	returns the lifespan of an object as a member of a given class
ref	$\mathcal{OI} \times \mathcal{TIME} \rightarrow 2^{\mathcal{OI}}$	returns the set of oids to which an object refers at a given instant
snapshot	$\mathcal{OI} \times \mathcal{TIME} \rightarrow \mathcal{V}$	projects the state of an object at a given instant

Table 3: Functions employed in defining the model

is denoted $list\text{-}of(T)$, where T is a type. Instances of $list\text{-}of(T)$ are lists of instances of type T. A record type is denoted $record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)$, where $T_1, \ldots, T_n$ are types and $a_1, \ldots, a_n \in \mathcal{AN}$ are distinct names. Instances of this type are records with n components, whose $i\text{-}th$ component is an instance of type T_i. Chimera allows the use of class names, i.e. object types, in the definition of structured types. Indeed, attributes of types structured as records, are allowed to have classes as domains. Moreover, the definition of collections, structured as sets or lists, of instances of classes must be supported in Chimera. The following definition introduces Chimera value types.

Definition 3.2 (Value Types) *[12] The set of Chimera value types $\mathcal{VT}$ is recursively 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-of(T) and set-of(T) are 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 elements of $\mathcal{AN}$, then record-of($a_1 : T_1, \ldots, a_n : T_n$) is a structured value type, indicated as record type.*

The set of Chimera types $\mathcal{CT}$ is defined as the union of Chimera value types $\mathcal{VT}$ and Chimera object types $\mathcal{OT}$.

T_Chimera extends the set of Chimera types with a collection of *temporal types*. For each Chimera type T, a corresponding temporal type, denoted as $temporal(T)$, is defined. First, the set

$\mathcal{BVT}$ of Chimera basic predefined value types is extended to contain also the type `time`. Intuitively, instances of type $temporal(T)$ are partial functions from instances of type `time` to instances of type T[2].

Definition 3.3 (T_Chimera Temporal Types) *The set of T_Chimera temporal types $\mathcal{TT}$ is defined as the set of types temporal(T), for each $T \in \mathcal{CT}$.*

In T_Chimera, temporal types can be used in the definition of structured types. The definition of sets or lists of instances of temporal types are allowed in T_Chimera. Moreover, attributes in a record are allowed to have temporal types as domain. The following definition formalizes these concepts.

Definition 3.4 (T_Chimera Types) *The set of T_Chimera types $\mathcal{T}$ is defined as follows.*

- `time` *is a T_Chimera type (*`time`$\in \mathcal{T}$*);*
- *the Chimera types are T_Chimera types ($\mathcal{CT} \subseteq \mathcal{T}$);*
- *the temporal types are T_Chimera types ($\mathcal{TT} \subseteq \mathcal{T}$);*
- *if T is a T_Chimera type then list-of(T) and set-of(T) are T_Chimera structured types;*
- *if $T_1, \ldots, T_n$ are T_Chimera types and $a_1, \ldots, a_n$ are distinct elements of $\mathcal{AN}$, then record-of($a_1 : T_1, \ldots, a_n : T_n$) is a record type.*

Example 3.1
Let `project` *be a class name, belonging to $\mathcal{CI}$. The following are T_Chimera types:*

```
time
temporal(integer)
list-of(boolean)
temporal(set-of(project))
record-of(task:temporal(project),startbudget:real,endbudget:real).
```

In the remainder of the discussion we make use of the following function: $\mathcal{T}^{-}$: $\mathcal{TT} \rightarrow \mathcal{CT}$, which takes as argument a temporal type $temporal(T)$ and returns the corresponding static type T. For example, $\mathcal{T}^{-}(temporal(\texttt{integer})) = \texttt{integer}$.

3.2 Values

In this subsection we introduce the set of T_Chimera legal values $\mathcal{V}$. For each T_Chimera type we introduce the corresponding set of legal values.

For each predefined basic value type $\mathcal{B} \in \mathcal{BVT}$, we postulate the existence of a non-empty set of values, denoted as $dom(\mathcal{B})$. For instance, the domain of the basic value type `real` is the set $\mathcal{R}$ of reals numbers, while the domain of the basic value type `bool` is the set $\{true, false\}$. Moreover, we assume as the domain of the type `time` the domain $\mathcal{TIME} = \{0, 1, \ldots, now, \ldots\}$, isomorphic to the set of natural numbers $\mathbb{N}$. Symbol '0' denotes the relative beginning, while *now* is a special constant denoting the current time. Thus, we assume time to be discrete. An interval, denoted as $I = [t_1, t_2]$ is a set of consecutive time instants. It includes all time instants between t_1 and t_2,

[2] We elaborate on this informal definition in the following section.

t_1 and t_2 included. A single time instant t can be represented as the time interval $[t, t]$, while $[]$ denotes the null interval, that is, the interval not containing any time instants. The operators of union ($I_1 \cup I_2$), intersection ($I_1 \cap I_2$) inclusion ($I_1 \subseteq I_2$) have the usual semantics of set operations. Moreover $t \in I$ is true if t is one of the time instants represented by interval I. In the following we use a set of disjoint intervals $I = \{[t_i, t_j], \ldots, [t_r, t_s]\}$ as a compact notation for the set of time instants included in these intervals.

Note that in T_Chimera oids in $\mathcal{OI}$ are handled as values. Thus, an object identifier i is a value of an object type in $\mathcal{OT}$. According to the usual terminology, an object is an instance of a class c, if c is the most specific class, in the inheritance hierarchy, to which the object belongs. If an object is an instance of a class it is also a member of all the superclasses of c. Therefore, we consider as legal values for an object type c all the oids of objects belonging to c both as instances or as members. The set of objects members or instances of a class changes dynamically over time. Thus, to define the extension, that is, the set of legal values for each T_Chimera type, we introduce a function π: $\mathcal{CI} \times \mathcal{TIME} \rightarrow 2^{\mathcal{OI}}$, assigning an extent to each class for each instant t. For each $c \in \mathcal{CI}$, $\pi(c, t)$ is the set of the identifiers of objects that, at time t, belonged to c both as instances or as members. To emphasize the fact that the interpretation of a type can only be given by fixing a time instant t, we denote the set of legal values for type T at time t as $[\![T]\!]_t$, meaning that this set can only be given fixing a time instant t.

Definition 3.5 (Type Legal Values) *$[\![T]\!]_t$ denotes the extension of type T at time t.*

- $null \in [\![T]\!]_t, \forall T \in \mathcal{T}$;
- $[\![\mathcal{B}]\!]_t = dom(\mathcal{B}), \forall B \in \mathcal{BVT}$;
- $[\![\texttt{time}]\!]_t = \mathcal{TIME}$;
- $[\![c]\!]_t = \pi(c,t), \forall c \in \mathcal{OT}$;
- $[\![set\text{-}of(T)]\!]_t = 2^{[\![T]\!]_t}$;
- $[\![list\text{-}of(T)]\!]_t = \{[v_1, \ldots, v_n] \mid n \geq 0, v_i \in [\![T]\!]_t, \forall i, 1 \leq i \leq n\}$;
- $[\![record\text{-}of(a_1 : T_1, \ldots, a_n : T_n)]\!]_t = \{(a_1 : v_1, \ldots, a_n : v_n) \mid a_i \in \mathcal{AN},\ v_i \in [\![T_i]\!]_t, \forall i,\ 1 \leq i \leq n\}$;
- $[\![temporal(T)]\!]_t = \{f \mid f\colon \mathcal{TIME} \rightarrow \bigcup_{t' \in \mathcal{TIME}} [\![T]\!]_{t'}$ *is a partial function such that* $\forall t'$ *if* $f(t')$ *is defined then* $f(t') \in [\![T]\!]_{t'}\}$.

Intuitively, given an instant t the extensions of predefined basic value types are the elements of their corresponding domains, the extensions of classes are their explicit extents at time t, while the set of legal values of the structured types are defined recursively in term of the legal values of their component types. The extension of a temporal type $temporal(T)$ is the set of partial functions from $\mathcal{TIME}$ (i.e, the set of legal values for type `time`) to the union of the set of legal values for type T for each instant t' in $\mathcal{TIME}$. The value of a variable of type $temporal(T)$ can then be represented as a set of pairs $(t, f(t))$, where f is a partial function, t is an element of $\mathcal{TIME}$ and $f(t)$ is the value of function f at time t. Usually, the value of a variable of temporal type does not change at each instant. Therefore, its value can be represented more efficiently as a set of pairs $\{\langle \tau_1, v_1 \rangle, \ldots, \langle \tau_n, v_n \rangle\}$, where $v_1, \ldots, v_n$ are legal values for the type T, and $\tau_1, \ldots, \tau_n$ are time intervals, such that the variable assumes the value v_i for each time instants in τ_i, $i = 1, \ldots, n$. We adopt this representation throughout the paper.

Example 3.2 *Let t be a time instant, i_1 and $i_2 \in \mathcal{OI}$ such that $i_1, i_2 \in \pi($*`person`*$,t)$ and $i_2 \in \pi($*`employee`*$,t)$.*

- $\texttt{10,100} \in [\![\texttt{integer}]\!]_t$;
- $\texttt{i}_2 \in [\![\texttt{employee}]\!]_t$;
- $\{\texttt{i}_1,\texttt{i}_2\} \in [\![\texttt{set-of(person)}]\!]_t$;
- $\{\langle\texttt{[5,10]},\texttt{12}\rangle,\langle\texttt{[11,30]},\texttt{5}\rangle\} \in [\![\texttt{temporal(integer)}]\!]_t$;
- $(\texttt{name:'Bob'}, \texttt{score:}\{\langle\texttt{[1,100]},\texttt{40}\rangle,\langle\texttt{[101,200]},\texttt{70}\rangle\}) \in$ $[\![\texttt{record}-\texttt{of}(\texttt{name}:\texttt{string},\texttt{score}:\texttt{temporal(integer)})]\!]_t$.

Definition 3.5 formally defines the set of legal values, that is, the extension for each T_Chimera type. In the following we introduce the corresponding typing rules.

Definition 3.6 (**Typing rules for values**). *The Chimera typing rules for values are the following*[3].

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

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

$$\frac{v \in \mathcal{TIME}}{v : \texttt{time}}$$

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

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

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

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

$$\frac{v_i : T, t_i : \texttt{time} \quad (1 \leq i \leq n)}{\{(t_1,v_1),\ldots,(t_n,v_n)\} : temporal(T)} \quad t_i \neq t_j, 1 \leq i,j \leq n$$

In the above rules, $\sqcup$ denotes the least upper bound of a set of types with respect to the subtyping order introduced in Section 6.

The above typing rules are used to check whether a database state is structurally consistent, that is, if the value of the attributes of each object meets the requirements of the structural part of its class definition[4]. Such typing rules are also the basis for type checking the expressions of T_Chimera language. The following theorems state the soundness and completeness of our type system.

Theorem 3.1 (Soundness of typing rules for values). *Let T be the type deduced for a value v according to rules in Definition 3.6, then there exists $t \in \mathcal{TIME}$ such that $v \in [\![T]\!]_t$.*

[3]The meaning of these inference rules is the following: if the conditions in the rule premises (the upper part of the rule) are satisfied, then the rule consequence (the lower part of the rule) can be inferred.

[4]We formalize these notions in the next section.

Theorem 3.2 (Completeness of typing rules for values). *Let v be a legal value for type T at time t, that is, $v \in [\![T]\!]_t$, then, according to rules in Definition 3.6, we deduce type T for v.*

The soundness can be easily proved by induction on the complexity of the derivation tree, whereas the completeness can be proved by induction on the structure of the type. We refer the interested reader to [4] for the proofs.

4 Classes

In this section we formally introduce the notion of class. A class in T_Chimera consists of two components: the *signature* and the *implementation.* The signature of a class contains all the information for the use of the class and its instances, whereas the implementation provides an implementation for the signature. The signature of a class contains information about the class identifier and the type of the class. A class can be *static* or *historical.* A class is static if all its c-attributes are static, that is, they do not have as domain a temporal type, it is historical otherwise. Moreover, a *lifespan* is associated with each class, representing the time interval during which the class has existed. As it does not make sense to recreate a class once it has been deleted, we make the assumption that the lifespan of a class is contiguous, that is, it consists of a set of consecutive time instants.

Furthermore, the class signature contains information about the attributes and methods of its instances. Each attribute is characterized by its name, and by the type of its values. Each method is characterized by its name, and by the type of the input and output parameters. Moreover, the signature contains similar information about c-attributes and c-methods. Since the objects belonging to a class vary over time, each T_Chimera class also maintains the history of all the objects instances or members of the class over time.

To model in a uniform way object features and class features we introduce the concept of *metaclass.* A metaclass is a special class having a class as unique instance. Each class is then seen as an instance of a metaclass in the same way as an object is seen as an instance of a class [10].

The signature of a class is formally defined as follows.

Definition 4.1 (Class Signature) *A class C is a 7-tuple*

`(c,type,lifespan,attr,meth,history,mc)`, *where*

`c` $\in \mathcal{CI}$ *is the class identifier;*

`type` $\in$ *{static,historical} indicates whether the class is historical, i.e. it contains at least one temporal c-attribute, or static;*

`lifespan` $\in (\mathcal{TIME} \times \mathcal{TIME})$ *is the lifespan of the class;*

`attr` *contains the information about the attributes of the class. It is a set containing an item for each attribute of the class. Such an item is a pair* `(a_name,a_type)`, *where*

`a_name` $\in \mathcal{AN}$ *is the attribute name;*

`a_type` $\in \mathcal{T}$ *is the attribute domain.*

`meth` *contains the information about the behavior of the instances of the class. It is a set containing an item for each method of the class. Each item is a pair* `(m_name,m_sign)` *where*

$\mathtt{m\_name} \in \mathcal{MN}$ *is the method name;*

$\mathtt{m\_sign}$ *is the signature of the method, expressed as:* $T_1 \times \ldots \times T_n \rightarrow T$*, where* $T_1, \ldots T_n$ *and* $T \in \mathcal{T}$*, denote respectively types of input parameters and of the output parameter of the method.*

`history` $\in \mathcal{V}$*, is a value containing the values for each c-attribute, plus two temporal values, representing respectively the objects instances of* C *and the objects members of* C *over time.* `History` *is a value of a record type, that is, it has the form*

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

where $a_1, \ldots, a_n$ *are the names of the c-attributes of* C*,* $v_1, \ldots, v_n$ *are their corresponding values,* E *and* PE *are temporal values representing the set of objects belonging to class* C *over time;*

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

The temporal attributes *ext* and *proper-ext* in the class history keep track of the objects members or instances of the class during its lifespan. They denote respectively, the set of the oids of objects members of the class for each instant t of its lifetime, and the oids of objects instances of the class. Obviously $PE(t) \subseteq E(t), \forall t \in C.lifespan$, because all objects instances of a class at a given instant are also members of the class at the same instant. Function π (cfr. Table 3), is such that, for each class name c', $\pi(c', t) = C.history.ext(t)$, for each $t \in C.lifespan$, where C is the (unique) class such that $C.c = c'$, that is, C is the class identified by c'.

Example 4.1 *Let us consider a class* `project`*, whose instances are research projects. Suppose moreover that objects of such a class have as attributes a* `name`*, which is immutable during the project lifetime, an* `objective` *and a* `workplan` *whose variations over time are not relevant for the application at hand, and a* `subproject` *and some* `participants`*, for which we are interested in keeping the whole history. The corresponding T_Chimera class signature is the following:*

$c = \mathtt{project}$
$type = \mathtt{static}$
$lifespan = [\mathtt{10}, \mathtt{now}]$
$attr = \{(\mathtt{name}, \mathtt{temporal(string)}), (\mathtt{objective}, \mathtt{string}), (\mathtt{workplan}, \mathtt{set\text{-}of(task)}),$
$\quad (\mathtt{subproject}, \mathtt{temporal(project)}), (\mathtt{participants}, \mathtt{temporal(set\text{-}of(person))}\}$
$meth = \{(\mathtt{add\text{-}participant}, \mathtt{person} \rightarrow \mathtt{project})\}$
$state = \mathtt{record\text{-}of(average\text{-}participants} : 20, ext : \{\langle[\mathtt{10}, \mathtt{now}], \{\mathtt{i}_1, \ldots, \mathtt{i}_4\}\rangle\},$
$\quad proper\text{-}ext : \{\langle[\mathtt{10}, \mathtt{50}], \{\mathtt{i}_1\}\rangle, \langle[\mathtt{51}, \mathtt{now}], \{\mathtt{i}_1, \mathtt{i}_2\}\rangle\})$
$mc = m$ - $\mathtt{project}$

The class `project` in the example above is a static class, since its only c-attribute `average-participants`, which records the current average numbers of project participants, is static. However, instances of class `project` are historical objects. If, by contrast, the c-attribute `average-participants` had recorded the changes of the average number of participants over time, the class would be historical.

Let us now discuss the relationships between a class and its associated types. 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 instances of C. Suppose that class C has as attr component the set: $\{(\mathtt{a_1},\mathtt{T_1}),\ldots,(\mathtt{a_n},\mathtt{T_n})\}$, the following types can be associated to C.

- *Structural type.* It represents the type of the attributes of instances of C. It is defined by the following record type:
  $\mathtt{record\text{-}of}(\mathtt{a_1}:\mathtt{T_1},\ldots,\mathtt{a_n}:\mathtt{T_n})$.
- *Historical type.* It represents the type of the temporal attributes of instances of C. It is defined by the following record type:
  $\mathtt{record\text{-}of}(\mathtt{a_k}:\mathtt{T'_k},\ldots,\mathtt{a_m}:\mathtt{T'_m})$
  let $\{(\mathtt{a_k},\mathtt{T_k}),\ldots,(\mathtt{a_m},\mathtt{T_m})\}, 1 \leq k \leq m \leq n$, be the subset of attr consisting of all the pairs $(\mathtt{a_i},\mathtt{T_i})$ such that $\mathtt{T_i}$ is a temporal type. Then $(\mathtt{a_i},\mathtt{T'_i})$ is such that $\mathtt{T'_i} = \mathcal{T}^{-}(\mathtt{T_i})$, $\forall i = k,\ldots,m$.
- *Static type.* It represents the type of the static attributes of instances of C. It is defined by the following record type:
  $\mathtt{record\text{-}of}(\mathtt{a_k}:\mathtt{T_k},\ldots,\mathtt{a_m}:\mathtt{T_m})$
  where $\{(\mathtt{a_k},\mathtt{T_k}),\ldots,(\mathtt{a_m},,\mathtt{T_m})\}, 1 \leq k \leq m \leq n$, is the subset of attr consisting of all the pairs $(\mathtt{a_i},\mathtt{T_i})$, such that $\mathtt{T_i}$ is not a temporal type.

The distinction between *structural*, *historical* and *static* type of a class will be used in the next section to check object consistency.

Therefore, we can define three functions *type*, *h_type*, *s_type*: $\mathcal{CI} \rightarrow \mathcal{T}$ which take as argument a class identifier c, and return the structural, the historical and the static type of the class identified by c, respectively[5].

Example 4.2 *Referring to the class of Example 4.1:*
h_type(`project`*)* = `record-of(name:string,subproject:project,participants:set-of(person))`
s_type(`project`*)* = `record-of(objective:string, workplan:set-of(task))`.

5 Objects

In this section we first introduce the notion of object, then we investigate the notions of object consistency and integrity in a temporal context. Finally, we consider object equality.

5.1 Objects

*T*_Chimera handles in a uniform way both historical and static objects. An object is historical if it contains at least one attribute with a temporal domain, it is static otherwise. With each object, either historical or static, a lifespan is associated, representing the time interval during which the object exists. As for classes we assume the lifespan of an object to be contiguous. Therefore, we do not consider a *reincarnate* operation like the one proposed in [7]. The contiguity assumption does not diminish the modeling power of our model. Objects can be instances of different classes during their lifetime, but we can assume that for each instant in their lifespan, there exists at least a class to which they belong. This class is the most general class (in the inheritance hierarchy) the object

[5]Note that function *h_type* returns a null value when its argument is the identifier of a class whose instances are static, whereas function *s_type* returns a null value when its input is a class whose instances only have temporal attributes.

has ever belonged to. For example, an employee can be fired and rehired, but he remains instance of the generic class `person`, superclass of the class `employee`, till the end of its lifetime.

Moreover, for each historical object the history of the most specific class to which it belongs during its lifespan is recorded. On the contrary, for each static object, only the class identifier of the most specific class to which it currently belongs is maintained. This information will be useful to check object consistency. The following definition formally introduces the notion of object.

Definition 5.1 (Object) *An object is a 4-tuple*

`(i,lifespan,v,class-history)` *where*

`i` $\in \mathcal{OI}$ *is the oid of o;*

`lifespan` $\in$ *(*$\mathcal{TIME} \times \mathcal{TIME}$*) is the lifespan of object o;*

`v` $\in \mathcal{V}$ *is a value, containing the values of each attribute of o. It is a value of record type:*
$(a_1 : v_1, \ldots, a_n : v_n)$
where $a_1, \ldots, a_n \in \mathcal{AN}$ *are the names of the attributes of o,* $v_1, \ldots, v_n$ *are their corresponding values;*

`class-history` $\in \mathcal{V}$ *is a value storing information about the most specific class to which o belongs over time. It is a temporal value:*
$\{\langle\tau_1, c_1\rangle, \ldots, \langle\tau_n, c_n\rangle\}$*, where* $\tau_1, \ldots, \tau_n$ *are time intervals and* $c_1, \ldots, c_n$ *are class identifier, such that* c_i *is the class identifier of the most specific class to which o belongs in the interval* τ_i*,* $1 \leq i \leq n$*.*

If o is static, `class-history` records only the most specific class to which o currently belongs. Thus `class-history` contains only one pair $\langle\tau, c\rangle$, where $\tau = [now, now]$ and c is the most specific class to which o belongs at the current time.

Example 5.1 *Suppose that* $i_1, i_2, i_3, i_4, i_7, i_8, i_9 \in \mathcal{OI}$ *and* `project` $\in \mathcal{CI}$*. The following is an example of T_Chimera object:*

$i = \mathtt{i}_1$
$lifespan = [20, \mathtt{now}]$
$attr\text{-}history = \{(\mathtt{name}, \{\langle[20, \mathtt{now}], '\mathtt{IDEA}'\rangle\}), (\mathtt{objective}, '\mathtt{Implementation}'),$
$(\mathtt{workplan}, \{\mathtt{i}_7\}),\ (\mathtt{subproject}, \{\langle[20, 45], \mathtt{i}_4\rangle, \langle[46, \mathtt{now}], \mathtt{i}_9\rangle\}),$
$(\mathtt{participants}, \{\langle[20, 80], \{\mathtt{i}_2, \mathtt{i}_3\}\rangle, \langle[81, \mathtt{now}], \{\mathtt{i}_2, \mathtt{i}_3, \mathtt{i}_8\}\rangle\}), \}.$
$class\text{-}history = \{\langle[20, \mathtt{now}], \mathtt{project}\rangle\}$

The object of the example above is an historical object, since it contains the temporal attributes `name`, `subproject` and `partecipants`. Thus, the *class-history* component keeps track of the most specific class to which the object belongs over time. In the considered example the object has never migrated, since it is an instance of the same class for all its lifespan.

In a temporal context, some temporal constraints must be satisfied by object lifespans. To formalize these constraints we define function $o\_lifespan$: $\mathcal{OI} \rightarrow \mathcal{TIME} \times \mathcal{TIME}$, that given an object oid i returns the lifespan of the object identified by i.

Obviously, information about the historical extent of a class must be consistent with the class history of objects in the database, as stated by the following invariant.

Invariant 5.1 *$\forall i \in \mathcal{OI}, \forall c' \in \mathcal{CI}, \forall t \in \mathcal{TIME}$, let o be the object such that $o.i = i'$, C be the class such that $c' = C.c$, then*

1. *$i' \in C.history.extent(t) \Rightarrow t \in o\_lifespan(i')$;*
2. *$(\forall t \in \tau, i' \in C.history.proper\text{-}extent(t)) \Leftrightarrow \langle \tau, c' \rangle \in o.class\text{-}history$.*

Moreover, the lifespan of an object can be partitioned in a set of intervals, depending on the object most specific class. Indeed, during its lifetime an object can be member of different classes. Therefore, we introduce function *c_lifespan*: $\mathcal{OI} \times \mathcal{CI} \rightarrow \mathcal{TIME} \times \mathcal{TIME}$, that given an object oid i and a class identifier c, returns the interval representing the set of time instants in which i was a member of the class identified by c[6].

The temporal constraints stated by the following invariant must be satisfied:

Invariant 5.2 *$\forall i \in \mathcal{OI}, \forall c' \in \mathcal{CI}, \forall t \in \mathcal{TIME}$, then*

1. *$o\_lifespan(i) = \bigcup_{c \in \mathcal{CI}} c\_lifespan(i, c)$;*
2. *$t \in c\_lifespan(i, c') \Leftrightarrow i \in C.history.extent(t)$, where C is the class such that $c' = C.c$* [7].

5.2 Consistency notions

The most specific class to which an object belongs can vary over time. An existing object can, at a certain point of its lifetime, be inserted in a more specific subclass or moved up to a more general superclass with respect to the one in which it is created. Moreover, an object can be an instance of the same class in different, not consecutive time instants. As an example, consider the case of an employee that is promoted to manager, (`manager` being a subclass of `employee` with some extra attributes, like `dependents` and `officialcar`). The other, rather undesirable case, is the transfer of the manager back to normal employee status (that means the loss of the official car and of the dependents). The migration of an object from a class to another can cause the addition or the deletion of some attributes from the object. With reference to the example above, the promotion of an employee to the manager status has the effect of adding the attributes `dependents` and `officialcar` to the corresponding object, while the transfer of the manager back to the employee status causes dropping the attributes `dependents` and `officialcar` from the corresponding object. If the attributes `dependents` and `officialcar` are are static, they are simply deleted from the object and no track of their existence is recorded in the object when it migrates to the class `employee`. By contrast, if they are temporal, the values they have assumed when the object has migrated to the class `manager` are maintained in the object, even if they are not part of the object anymore.

We require that each object must be a consistent instance of all the classes to which it belongs. In a context where objects can have both static and temporal attributes, the notion of consistency assumes a slightly different semantics with respect to its classical definition. Verifying the consistency of an object in a temporal context requires two steps. First, it is necessary to identify for each instant t of the object lifespan, the set of attributes which characterize the object at time t, as the attributes characterizing a given object can vary over time. Then, the correctness of their values must be checked. Note that, if we consider a time instant t lesser than the current time,

[6] Note that $c\_lifespan(i, c) = \bigcup_{\langle \tau_i, c_i \rangle \in\, o.class-history,\, c_i\ subclass\ of\ c} \tau_i$. Functions o_lifespan and c_lifespan are similar to those defined in [21].

[7] This also implies that $t \in c\_lifespan(i, c') \Leftrightarrow i \in [\![c']\!]_t$.

we are able to identify only the temporal attributes that characterize the object at time t, since for static attributes we record only their current values. Thus, for instants lesser than the current time, it only makes sense to check the correctness of the values of the temporal attributes of the objects. Therefore, we start by introducing the following definition:

Definition 5.2 (Meaningful Temporal Attributes) *Let a be a temporal attribute of an object o. Let v be its value. Attribute a is said to be meaningful for o at time t, if t belongs to the domain of v.*

Note that, as the value of a temporal attribute is a partial function from $\mathcal{TIME}$ to the attribute value domain, its domain is the set of time instants for which the value is defined. Therefore, the domain of the value of a temporal attribute of an object o represents the set of time instants in which the attribute characterizes o.

We distinguish two kinds of consistency:

- *Historical consistency.* The values of the temporal attributes of the object at a given instant are legal values for the temporal attributes of the class.
- *Static consistency.* The values of the static attributes of the object are legal values for the static attributes of the class.

Consider an object $o = (i, lifespan, v, class\text{-}history)$, such that $v = (\mathtt{a_1} : \mathtt{v_1}, \ldots, \mathtt{a_n} : \mathtt{v_n})$. Therefore, given an instant $t \in o.lifespan$, the following values can be defined:

- *Historical value.* It is a record representing the values of the temporal attributes meaningful for the object at time t. Let $\{\mathtt{a}_k,\ \ldots, \mathtt{a}_m\}$, $1 \leq k \leq m \leq n$, be the subset of $\{\mathtt{a}_1,\ \ldots, \mathtt{a}_n\}$, consisting of all the names of the temporal attributes meaningful for o at time t. The historical value of o at time t is defined as follows:
  $(\mathtt{a_k} : \mathtt{v_k(t)}, \ldots, \mathtt{a_m} : \mathtt{v_m(t)})$
  where $\mathtt{v_i(t)}$ denotes the value of $\mathtt{a_i}$ at time t, $i = k, \ldots, m$.
- *Static value.* It is a record representing the values of the static attributes of the object. Its definition is analogous to that of the historical value, considering static attributes instead of the temporal ones.

Thus, we can define two functions *h_state*: $\mathcal{OI} \times \mathcal{TIME} \rightarrow \mathcal{V}$ which takes as argument an object identifier and a time instant t, and returns the record representing the historical value of the object at time t, *s_state*: $\mathcal{OI} \rightarrow \mathcal{V}$, which takes as argument an object identifier, and returns the static value of the object. Note that when an object consists only of temporal attributes, function *h_state* returns a snapshot of the value of the object attributes for a specified time instant.

Example 5.2 *Referring to object of Example 5.1:*
$s\_state(\mathtt{i_1}) = (\mathtt{objective} :' \mathtt{Implementation}', \mathtt{workplan} : \{\mathtt{i_7}\})$
$h\_state(\mathtt{i_1}, 50) = (\mathtt{name} :' \mathtt{IDEA}', \mathtt{subproject} : \mathtt{i_9}, \mathtt{participants} : \{\mathtt{i_2}, \mathtt{i_3}\})$

We are now ready to formally introduce the notions of *historical* and *static* consistency, by making use of functions *h_type* and *s_type* (cfr. Table 3).

Definition 5.3 (Historical Consistency) *An object o = (i,lifespan,v,class-history) is an historically consistent instance of a class c' at time t if h_state(o.i,t) is a legal value for the type h_type(c').*

Definition 5.4 (Static Consistency) *An object o = (i,lifespan,v,class-history) is a statically consistent instance of a class c', if s_state(o.i) is a legal value for the type s_type(c').*

Checking the consistency of an historical object means verifying its consistency with all the classes to which it belongs during its lifespan. For time instants lesser than the current time, only its historical consistency can be checked, since only for temporal values we have the history of all their changes. For the current time also the static consistency must be checked. The consistency of an object is checked only with respect to its most specific class. If an object is consistent with respect to its most specific class, it is consistent with respect to all the superclasses.

Definition 5.5 (Object Consistency) *An object o = (i,lifespan,v,class-history) is consistent if and only if the following conditions hold:*

- *For each pair $\langle \tau, c' \rangle$ in o.class-history, interval τ is contained in the lifespan of the class identified by c', that is, $\tau \subseteq C.lifespan$, where C is the class such that $C.c = c'$.*
- *For each pair $\langle \tau, c' \rangle$ in o.class-history, o is an historical consistent instance of c', for each instant $t \in \tau$.*
- *Let $\langle \tau, c \rangle$ be the (unique) element of o.class-history, such that* now $\in \tau$. *Object o must be a static consistent instance of class c.*

The above definition states that each object, for each instant t of its lifespan, must contain a value for each temporal attribute of the class to which it belongs at time t, and this value must be of the correct type. Moreover, at the current time also the consistency with respect to the static attributes must be checked. This notion of consistency allows to uniformly treat both static and historical objects. In the case of static objects, Definition 5.5 reduces to the notion of consistency usually adopted in traditional object-oriented system [14].

Example 5.3 *The object of Example 5.1 is a consistent instance of the class of Example 4.1 provided that*

- $\mathtt{i}_7 \in [\![\mathtt{task}]\!]_{now}$;
- $\mathtt{i}_4 \in [\![\mathtt{project}]\!]_t$, *for each* $t \in [20, 45]$;
- $\mathtt{i}_9 \in [\![\mathtt{project}]\!]_t$, *for each* $t \in [46, now]$;
- $\mathtt{i}_2, \mathtt{i}_3 \in [\![\mathtt{person}]\!]_t$, *for each* $t \in [20, now]$;
- $\mathtt{i}_8 \in [\![\mathtt{person}]\!]_t$, *for each* $t \in [81, now]$.

Objects can be defined by means of a set of component objects. An object o is said to *refer to* an object o' at an instant t, if the object identifier of o' appears in one of its attribute values at time t. In traditional object-oriented system, the *referential integrity* property must be enforced, that is, if an object is in the set, also all the objects to which it refers must be in the set. Analogously in our temporal model we require that if at instant t an object o refers an object o', both o and o' exist, that is, instant t belongs to the lifespan of both o and o'. Moreover, the integrity of objects identity is enforced through object oids. The oid of an object represents a time invariant property of the object, shared by no other object in the system. It is analogous to the concept of *essence* introduced in [7]. Thus, for a set of objects to be consistent the property of *oid-uniqueness* must be ensured. These notions are formalized by Definition 5.6.

In the following definition we make use of function ref that, receiving in input an object identifier i and a time instant t, returns the set of identifiers of the objects to which the object identified by i refers at time t (cfr. Table 3). Moreover, given a set of objects OBJ, I(OBJ) denotes the set $\{i' \mid i' = o.i, o \in \text{OBJ}\}$.

Definition 5.6 (Consistent Set of Objects) *A finite set of objects* OBJ *is consistent at time t if and only if the following conditions hold:*

1. OID-UNIQUENESS
   For each pair of objects o_1 and $o_2 \in$ OBJ, if $o_1.i = o_2.i$, then o_1.lifespan = o_2.lifespan, $o_1.v$ = $o_2.v$ and o_1.class-history = o_2.class-history;

2. REFERENTIAL INTEGRITY
   *for all objects $o \in$ OBJ, each object identifier in ref(o.i,t) must be contained in I(*OBJ*).*

5.3 Object equality

Object equality is a fundamental concept in object-oriented languages. Chimera, like most object-oriented systems, supports two different notions of equality: *equality by identity* (meaning that the two denoted objects are the same object), and *equality by value* (meaning that the two objects have the same attribute values). Equality by value can be further refined into: *shallow value equality*, which considers the equality of all the direct attributes of an object, and *deep value equality*, which considers in addition to the equality of the attributes the equality of the attributes of objects which are recursively reached by means of oid references. A formalization of these concepts for the Chimera model can be found in [12]. Here, we consider only shallow value equality.

In a temporal context, we still have the classical notion of object identity, being the oid of an object a time invariant property. This notion is formalized as follows.

Definition 5.7 (Equality by Identity) *Two objects o_1 and o_2 are equal by identity if and only if they have the same object identifier, that is, $o_1.i$=$o_2.i$.*

Obviously, if two objects have the same oid, also all their other components are equal, because of the OID-UNIQUENESS property. Note, moreover, that the notion of object identity uniformly applies both to historical and to static objects.

Value equality can be formally defined as follows.

Definition 5.8 (Value Equality) *Two objects o_1 and o_2 are equal by value if and only if $o_1.v = o_2.v$.*

Note that the above definition implies both the equality of the attribute values and the attribute names. Definition 5.8 applies both to static and historical objects. In the case of static objects, it simply reduces to the conventional notion of value equality, whereas for historical objects the equality of the whole history of temporal attributes is required.

In a temporal object-oriented context, two further notions of value equality can be devised: *instantaneous-value equality* and *weak-value equality*. Two objects are instantaneously value equal if there exists an instant t in which their attributes have the same values. Two objects are said to be weakly value equal if their attributes have ever had the same values, also in different instants. These notions of equality obviously make sense for historical objects consisting only of temporal attributes. Moreover, also historical objects containing static attributes can be compared under these types of equalities, but only at the current time, since we cannot reconstruct the value of static attributes at a time instant t lesser than the current time. Finally, static objects can be compared only at the current time too.

Let us consider a function *snapshot* : $\mathcal{OI} \times \mathcal{TIME} \rightarrow \mathcal{V}$ that given an object identifier i' and an instant t "projects" the state of the object identified by i' at time t. This function returns a record value $(a_1 : v_1, \ldots, a_n : v_n)$ such that, for all j, $1 \leq j \leq n$,

- if attribute a_j is static, $v_j = o.v.a_j$,
- if a_j is temporal, $v_j = o.v.a_j(t)$,

being o the object identified by i' $(o.i = i')$[8]. For historical objects containing also static attributes, we are able to reconstruct only the snapshot at the current time, thus $snapshot(i, t)$, with i identifier of an historical object with at least a static attribute, is undefined for $t \neq now$. Note that, as a particular case, the snapshot of a static object at any instant is its current state. For instance, referring to the object of Example 5.1, $snapshot(\mathtt{i}_1, now) = (\mathtt{name} : \text{'}\mathtt{IDEA}\text{'}$, $\mathtt{objective} : \text{'}\mathtt{Implementation}\text{'}, \mathtt{workplan} : \{\mathtt{i}_7\}, \mathtt{subproject} : \mathtt{i}_9, \mathtt{participants} : \{\mathtt{i}_2, \mathtt{i}_3, \mathtt{i}_8\}$, whereas $snapshot(\mathtt{i}_1, now)$, for $t \neq now$ is undefined.

Definition 5.9 (Instantaneous-value Equality) *Two objects o_1 and o_2 are instantaneously value equal if there exists an instant $t \in o_1.lifespan \cap o_2.lifespan$ such that, $snapshot(o_1.i,t) = snapshot(o_2.i,t)$.*

Definition 5.10 (Weak-value Equality) *Two objects o_1 and o_2 are weakly value equal if there exist two instants $t' \in o_1.lifespan$ and $t'' \in o_2.lifespan$, such that $snapshot(o_1.i,t') = snapshot(o_2.i,t'')$.*

Note that the last three notions of equality do not require that the two objects are instances of classes related in the inheritance hierarchy. Obviously, value equality implies instantaneous-value equality and instantaneous-value equality implies weak-value equality, while equality by identity implies all the other kinds of equality, when they are applicable.

The usefulness of these notions of equality is exemplified by the following example.

Example 5.4 *Referring to the class* `project` *of Example 4.1, two* `project` *objects having the same current state and the same history of modifications for* `subproject` *and* `participants` *attributes, are value equal. By contrast, two* `project` *objects having the same current value for all the attributes are instantaneous (and thus, weak) value equal.*

6 Inheritance

Inheritance relationships among classes are described by an ISA hierarchy established by the user. The ISA hierarchy represents which classes are subclasses of (inherit from) other classes. We can suppose this information to be expressed as a partial order $\leq_{ISA}$ on the set of class identifiers $\mathcal{CI}$.

[8] For historical objects containing only temporal attributes, the *snapshot* function coincides with the *h_state* function (cfr. Table 3); that is, if i is the identifier of an object of this kind, for any instant t $snapshot(i, t) = h\_state(i, t)$ holds.

Inheritance has two important implications in Chimera, like in other object-oriented data models in which classes are associated with an extent. The first one is *substitutability*, that is, the property that each instance of a class can be used whenever an instance of one of its superclasses is expected. The second one is *extent inclusion*, that is, the property ensuring that the extent of a class is included in that of its superclasses. In what follows we consider these two aspects.

6.1 Substitutability

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 and operations (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. We also require that the extent of a subclass is a subset of the extent of all its superclasses.

To formally define notions such as domain refinement, we need to define an ordering on types. The subtype relationship $\leq_T$ is defined as follows.

Definition 6.1 (Subtypes) *Given $T_1, T_2 \in \mathcal{T}$, T_2 is a subtype of T_1 (denoted as $T_2 \leq_T T_1$) iff at least one of the following conditions holds:*

- $T_1 = T_2$;
- $T_2, T_1 \in \mathcal{OT}$ *and* $T_1 \leq_{ISA} (T_2)$;
- $T_2 = set\text{-}of(T_2')$, $T_1 = set\text{-}of(T_1')$ *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, $1 \leq i \leq n$, $T_i' \leq_T T_i''$;
- $T_2 = temporal(T_2')$, $T_1 = temporal(T_1')$ *and* $T_2' \leq_T T_1'$.

The set $\mathcal{T}$ of types with the ordering $\leq_T$ is a poset. Thus, the notion of least upper bound ($\sqcup$) is well defined (cfr. Subsection 3.2).

Given the above definition of subtyping, the following rule establishes refinement conditions for attribute domains in subclasses.

Rule 6.1 (Refinement of attribute domains) *Let c_1 be a class, and a an attribute whose domain in c_1 is T. Let c_2 be a class such that $c_2 \leq_{ISA} c_1$, attribute a in class c_2 can have as domain a type $T' \in \mathcal{T}$ such that:*

1. $T' \leq_T T$, *or*
2. $T' = temporal(T'')$, $T'' \leq_T T$.

This means that, as suggested in [6], a non-temporal attribute can be refined in a temporal attribute (on the same domain or on a most specific one), but not vice-versa.

To ensure substitutability, however, we need to introduce a *coercion* function, since the value of a temporal attribute, which is a function from a temporal domain, cannot be substituted (from a typing viewpoint) by a value of a non-temporal attribute, which is not a function. Suppose that class c_2 refines the non-temporal domain of attribute a of its superclass c_1 in a temporal domain. Then, whenever an object o, instance of c_2, must be seen as an instance of c_1, the value of attribute a is "coerced" to $o.v.a(now)$, that is, the value of the function which is the value of the temporal attribute a at the instant *now*. The coercion function we make use of is the *snapshot* function (cfr. Table 3). By considering a snapshot of the object at the instant *now*, the coerced value is $snapshot(i, now).a$. Note that this coercion is semantically meaningful. Indeed, in c_1 we are not interested in the history of the values taken by attribute a. Thus, whenever we see an object instance of c_2 as an instance of c_1, we forget the history of attribute a and consider only its current value.

6.2 Extent inclusion

At the extensional level, we require that in each database state the extent of a class is a subset of the extent of its superclasses, as stated by the following invariant.

Invariant 6.1 *Given* $c_1, c_2 \in \mathcal{CI}$, *such that* $c_2 \leq_{ISA} c_1$, *and* $c_1 = C_1.c$, $c_2 = C_2.c$, *then*

1. $C_2.lifespan \subseteq C_1.lifespan$;
2. $\forall t \in C_2.lifespan$, $C_2.history.ext(t) \subseteq C_1.history.ext(t)$;
3. $\forall i \in \mathcal{OI}$ $c\_lifespan(i, c_2) \subseteq c\_lifespan(i, c_1)$.

It is important to note that in Chimera a common superclass of all the classes does not exist. Therefore the hierarchy is a DAG, consisting of a number of connected components whose roots are the classes without superclasses, which we call *root classes*. 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 $C_1, \ldots, C_m$ and a time instant $t \in \mathcal{TIME}$, then Ext_i^t, $i = 1, \ldots, m$, denotes the extension of C_i at time t (that is, $C_i.history.extent(t)$) which is the extent of the entire hierarchy rooted at C_i at this time. The following invariant must hold, stating that the sets of objects that have ever belonged to different hierarchies are disjoint (that is, an object cannot belong to different hierarchies even at different times), since an object cannot migrate over different hierarchies.

Invariant 6.2 *Let* $C_1, \ldots, C_m$ *be the root classes of the ISA relationship, then for each* i, j *with* $i \neq j$, $1 \leq i, j \leq m$,

$$\bigcup_{t \in \mathcal{TIME}} Ext_i^t \cap \bigcup_{t \in \mathcal{TIME}} Ext_j^t = \emptyset$$

The invariant also implies that at each time the sets of objects belonging to different hierarchies are disjoint.

Finally, the following result holds, relating the subtype relationship to type extensions defined in Section 3.

Theorem 6.1 *If* $T_1 \leq_T T_2$, *then* $\forall t \in \mathcal{TIME}$, $[\![T_1]\!]_t \subseteq [\![T_2]\!]_t$ *holds.*

7 Conclusion and future work

In this paper we have presented T_Chimera, an extension of the Chimera data model incorporating temporal capabilities. We have introduced the notion of temporal type to handle in a uniform way temporal and non-temporal domains, and we have defined the set of legal T_Chimera values for each type. We have discussed the notion of object consistency and integrity and we have also investigated problem related to inheritance and object identity in a temporal framework.

We plan to extend this work in several directions. First, we plan to extend Chimera triggers and deductive rules with time. In particular, temporal triggers have not been much investigated; issues such as termination and confluence will need to be re-visited when dealing with temporal triggers. Second, we plan to define a temporal integrity constraint language for the Chimera data model. Such a language would allow, among other things, to express constraints based on past histories of objects. Then, we are interested in investigating temporal object references and, more generally, issues related to the query language and its typing. Time-dependent behavior is an interesting topic of future work, too. Finally, implementation issues will be investigated.

References

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

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

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

[4] E. Bertino, E. Ferrari and G. Guerrini. A Formal Temporal Object-Oriented Data Model. Technical Report, University of Milano, May 1995. (Extended Version of this paper).

[5] 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, pages 1–15, 1988.

[6] T. Cheng and S. Gadia. An Object-Oriented Model for Temporal Databases. In *Proc. of the Int'l Workshop on an Infrastructure for Temporal Databases*, 1993.

[7] J. Clifford and A. Croker. Objects in Time. In *Proc. Fourth IEEE Int'l Conf. on Data Engineering*, pages 11–18, 1988.

[8] A. Crocker and J. Clifford. The Historical Relational Data Model (HRDM) Revisited. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 6–26. Benjamin/Cummings, 1993.

[9] S.K. Gadia and S.S. Nair. Temporal Databases: A Prelude to Parametric Data. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 28–66. Benjamin/Cummings, 1993.

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

[11] I. Goralwalla and M. Özsu. Temporal Extensions to a Uniform Behavioral Object Model. In R. Elmasri, V. Kouramajian, and B. Thalheim, editors, *Proc. Twelfth Int'l Conf. on the Entity-Relationship Approach*, volume 823 of *Lecture Notes in Computer Science*, pages 110–121. Springer-Verlag, Berlin, 1993.

[12] G. Guerrini, E. Bertino, and R. Bal. A Formal Definition of the Chimera Object-Oriented Data Model. Technical Report IDEA.DE.2P.011.01, ESPRIT Project 6333, May 1994. Submitted for publication.

[13] W. Käfer and H. Schöning. Realizing a Temporal Complex-Object Data Model. In M. Stonebraker, editor, *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 266–275. ACM Press, 1992.

[14] 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.

[15] N. Pissinou and K. Makki. A Framework for Temporal Object Databases. In *Proc. First Int'l Conf. on Information and Knowledge Management*, volume 752 of *Lecture Notes in Computer Science*, pages 86–97. Springer-Verlag, Berlin, 1992.

[16] R. Snodgrass. The Temporal Query Language TQUEL. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[17] R. Snodgrass. Temporal Object-Oriented Databases: A Critical Comparison. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability and Beyond.* Addison-Wesley/ACM Press, 1995.

[18] R. Stam and R. Snodgrass. A Bibliography on Temporal Databases. Database Engineering, December 1988.

[19] S. Su and H. Chen. A Temporal Knowledge Representation Model OSAM*/T and its Query Language OQL/T. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. Seventeenth Int'l Conf. on Very Large Data Bases*, pages 431–441, 1991.

[20] A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases: Theory, Design, and Implementation.* Database Systems and Applications Series. Benjamin/Cummings, 1993.

[21] G. Wuu and U. Dayal. A Uniform Model for Temporal and Versioned Object-Oriented Databases. In A. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, editors, *Temporal Databases: Theory, Design, and Implementation*, pages 230–247. Benjamin/Cummings, 1993.

[22] S. Zdonik. Object-Oriented Type Evolution. In F. Bancilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 277–288. Addison-Wesley, 1990.