Attribute Refinement in a Multigranular Temporal Object Data Model *

E. Bertino[1] E. Camossi[2] G. Guerrini[3]

[1] CERIAS - Purdue University, 250 N. University Street West Lafayette, Indiana, USA 47907-2066. Phone: +1 765 496-2399. Fax: +1 765 494-0739. e-mail: bertino@cs.purdue.edu
[2] School of Computer Science and Informatics - University College Dublin, Belfield, Dublin 4, Ireland. Phone: +353 (0)1 7162-944. Fax: +353 (0)1 2697-262. e-mail: {elena.camossi}@ucd.ie
[3] DISI - Università degli Studi di Genova, Via Dodecaneso 35, 16146 Genova, Italy. Phone: +39 010 353-6701. Fax:+39 010 353-6699. e-mail: guerrini@disi.unige.it

Abstract. Temporal granularities are the unit of measure for temporal data, thus a multigranular temporal object model allows to store temporal data at different levels of detail, according to the needs of the application domain. In this paper we investigate how the integration of multiple temporal granularities in an object-oriented data model impacts on the inheritance hierarchy. In the paper we specifically address issues related to attribute refinement, and the consequences on object substitutability. This entails the development of suitable instruments for converting temporal values from a granularity to another.

1 Introduction

Temporal object data models allow to maintain the values taken by object attributes over time. Conventional object database systems do not offer a support for dealing with time-varying objects. The content of a database represents a *snapshot* of the reality in that only the current values of object attributes are recorded, without the possibility of maintaining the complete history of objects over time. If such a need arises, object attribute histories must be managed at application program level. A direct support for temporal objects at database level, by contrast, would greatly simplify their management and handling. Thus, in the past years, there has been a growing interest in temporal extensions to the current database technology. Several extensions to the relational and the object-oriented data models and query languages have been proposed [15, 24].

An important requirement, when dealing with temporal aspects, concerns the support for *multiple temporal granularities* [7]. Temporal granularities are the

* Research presented in this paper was funded by a Strategic Research Cluster grant (07/SRC/I1168) by Science Foundation Ireland under the National Development Plan. The authors gratefully acknowledge this support. The work of Elena Camossi is supported by the Irish Research Council for Science, Engineering and Technology.

unit of measure for temporal data. For instance, birth dates are typically referred to the granularity of days and train schedules to that of minutes. The choice of the correct temporal granularity allows the system to store the minimal amount of data, according to the needed level of detail. Since many different granularities exist and no granularity is inherently "better" than another, a temporal database system should support a wide range of temporal granularities and should allow the user to define his/her own application-specific granularities.

Temporal relational databases, and in particular the TSQL2 [21] extension of the SQL-92 standard, support multiple temporal granularities. Some proposals of temporal object models supporting multiple temporal granularities also exist, but they suffer for lack of standardization, lack of formalization, and they mostly leave the user the burden of managing granularities. The introduction of temporal granularities in object-oriented data models, moreover, raises many relevant issues, that are not addressed by existing proposals. For instance, none of the existing approaches considers the impact of inheritance when multiple granularities are supported, nor they address issues concerning type refinement and substitutability. We strongly believe that systems must provide a support for handling multiple granularities, and that all the notions of a temporal object model should be revised for adequately supporting multiple granularities. The modelling power of an object data model combined with a multigranular representation for temporal data will result in a highly flexible data model, suitable for all applications needing the management of temporal information.

In [5] we proposed T_ODMG, a temporal object model supporting multiple temporal granularities. To overcome the lack of standardization of current proposals, the model is defined as an extension of the ODMG object database standard data model [11]. Though most commercial OODMSs still do not exhibit a full level of compliance to the ODMG standard, we think that casting our temporal extension within the ODMG standard makes it more understandable and easily adoptable by commercial systems. Moreover, since the ODMG Java binding is the basis on which Java Data Objects [23] has been developed, our proposal can also be easily applied to persistent Java applications.

T_ODMG objects are characterized by temporal properties, each one referring to a specific granularity. A temporal property referring to granularity G can assume a different value v on each granule g of granularity G. For instance, a course object may have a temporal property recording the teacher teaching the course each year; a city object may have a temporal property recording the city temperature every month; finally, an employee object may have a temporal property recording the salary earned by the employee each year.

An important assumption made in [5] is that properties are *downward inheritable* [20]. This means that if a temporal property assumes value v on a granule g, value v also refers to any granule g' of a finer granularity included in g. Thus, for instance, if we choose to record course teachers at the year granularity, this implies that the teacher is associated with each month (and each day, ...) of the year. This assumption, however, is not reasonable for all kinds of properties. Consider the temporal value representing the temperature of a city, at the month

granularity. It is not always the case that the temperature in the city was the same for every day of the month, though the monthly temperature is probably a reasonable approximation of the (non available) daily values. Even more evident, consider now the temporal value representing the salary of an employee, at the year granularity. It is not always the case that the salary of the employee was the same for every month of the year, and the yearly salary is probably not a reasonable approximation of the (non available) monthly values.

In this paper, we remove this assumption and we revisit the notion of attribute refinement along the inheritance hierarchy taking multiple granularities into account, and investigate the impact of such refinements on substitutability. Substitutability ensures that each instance of a given class can be used whenever an instance of one of its superclasses is expected. The idea behind attribute refinement is that the granularity at which an attribute value is stored can be changed in a subclass, to better reflect the application needs. In the subclass the attribute values may should be kept at a greater level of detail. For instance, if in the superclass only the monthly values are recorded, in the subclass the daily changes can be of interest. Depending on the application domain and on the attribute semantics, an attribute value may should be recorded at a decreased level of detail in the subclass. For instance, if in the superclass the monthly values are recorded, in the subclass only the yearly values can be stored.

Attribute refinement thus means changing the level of detail at which attribute values are stored. Attribute refinement impacts substitutability, since whenever an object instance of a subclass is found in a context where a superclass object was expected, the problem arises of *converting* its temporal attribute values to the expected granularity. Such a conversion is needed both for attribute accesses and for attribute updates. To address this issue, we introduce the notions of *coercion* and *refinement* functions. These functions are used, in case of object access, to compute the value to be considered in the superclass, given the value of the attribute in the subclass, and, in case of object update, to convert the value to assign to the granularity required in the subclass.

The need of converting temporal values from one granularity to another arises in several contexts and it is not only related to enforcing substitutability in case of attribute refinement. Besides for generic casts, such a conversion is needed for supporting full-fledged query languages, in which temporal values expressed at different granularities can be used together, as well as the basis of any schema evolution mechanism, allowing the granularity of a temporal property to dynamically change, with an automatic adaptation of previous property values. Thus, coercion and refinement functions are important notions for multigranular temporal value handling, interesting also independently from attribute refinement.

Thus, the contributions of this paper can be summarized as follows. A temporal multigranular object data model is defined, in which temporal types are modeled as parametric types [9]. The notions of coercion and refinement functions, allowing to convert temporal values from a granularity to another, are introduced, and some relevant properties of such functions are devised. Attribute refinement, for attributes whose domain is a temporal type, is then investigated,

discussing its impact on substitutability, and relying on coercion and refinement functions to convert attribute values to the expected granularity.

The paper is organized as follows. Section 2 introduces the notion of temporal granularities we refer to. Section 3 introduces the temporal object data model, while Section 4 is specifically devoted to attribute refinement in the model. Section 5 presents how the proposed mechanism has been implemented, Section 6 surveys related work, while Section 7 concludes the paper.

2 Temporal Granularities

The notion of time we refer to is *valid time*, that is, we record the time at which a given value is taken in reality, in contrast to *transaction time*, corresponding to when a given value is recorded in the database [14]. Our approach can however be easily generalized to bitemporal models, supporting both transaction and valid times. The time domain is the pair ($\mathbb{N}$,$\leq$), where $\leq$ is the usual less than or equal to relation on $\mathbb{N}$. Thus, we consider a discrete time domain and we assume the existence of a relative beginning, denoted by symbol "0", but no last element. Moreover, we consider a special symbol ∞ denoting the distinguished time instant $forever$ [14]. Temporal granularities are formally defined as follows.

Definition 1. *(*Granularity*)[6]. Let $\mathcal{IS}$ be an index set isomorphic to the set of natural numbers $\mathbb{N}$[1], and $2^{\mathbb{N}}$ be the power set of the time domain. A* granularity *G is a mapping from $\mathcal{IS}$ to $2^{\mathbb{N}}$ such that both the following conditions hold:*

(1) if $i < j$ and $G(i)$ and $G(j)$ are non-empty, then each element of $G(i)$ is less than all elements of $G(j)$;
(2) if $i < k < j$ and $G(i)$ and $G(j)$ are non-empty, then $G(k)$ is non-empty. □

Intuitively a granularity defines a countable set of *granules*, such that each granule $G(i)$ is associated with an index $i \in \mathcal{IS}$ and denotes a subset of the time domain. The first condition in Definition 1 states that granules in a granularity do not overlap and that their index order is the same as their time domain order. The second condition states that the subset of the index set that maps to non-empty subsets of the time domain is contiguous. The set of granularities we refer to throughout the paper is denoted by $\mathcal{G}$.

The usual collections *days*, *months*, *weeks* and *years* are granularities. For each non-empty granule, we use a "textual representation", termed as *label*. For example, *days* are in the form *mm/dd/yyyy*, *months* are in the form *mm/yyyy* and so on. When we refer to a generic granularity G and a granule index $i \in \mathcal{IS}$, l_i^G denotes the label of the granule corresponding to the ith granule of granularity G. A label is more descriptive than a granule index. Indeed, given a granule label

[1] Though $\mathcal{IS}$ is isomorphic to $\mathbb{N}$, the two concepts of time domain and index set are clearly different. Thus, they will be denoted by different symbols: $\mathbb{N}$ and $\mathcal{IS}$, respectively. Moreover, generic indexes will be denoted by i, j, k whereas generic time instants will be denoted by t, t', t_1.

l_i^G, we know which is the granularity (G) and which is the index (i) of the denoted granule.

A particular case of granularities is represented by *gap granularities* [6], that is, granularities whose domain does not correspond to the whole time domain. A typical example of gap granularity is granularity *bweeks*, corresponding to business weeks, that is, to sequences of the five working days from Monday to Friday. There are portions of the time domains, e.g., those corresponding to Sundays, that do not belong to any granule of that granularity.

A *finer than* relationship is defined among granularities, as follows.

Definition 2. *(*Finer than Relationship*)[6]. A granularity $G \in \mathcal{G}$ is said to be* finer than *a granularity H, denoted $G \preceq H$, if, for each index $i \in \mathcal{IS}$, an index $j \in \mathcal{IS}$ exists such that $G(i) \subseteq H(j)$.* □

Note that $G \preceq G$ is always true according to the above definition. In what follows the symbol "$\prec$" denotes the anti-reflexive finer than relationship. As an example, *days* is a granularity finer than *months* ($days \preceq months$). As another example, consider the gap granularity *bweeks* introduced above. Since each *bweeks* granule is included in a *weeks* granule, $bweeks \preceq weeks$ holds. By contrast, since not every *days* granule is included in a *bweeks* granule (e.g., granules corresponding to Sundays), *days* is not finer than *bweeks*.

A set of granularities having the same time domain forms a *granularity lattice* with respect to the $\preceq$ relationship if, for each pair of granularities in the set, a least upper bound (*lub*) and a greatest lower bound (*glb*) with respect to $\preceq$ exist. It is easy to see that this does not hold for very common sets of granularities. For instance, the granularity set $\{weeks, months\}$ does not have either a lub or a glb with respect to $\preceq$. The set of granularities $\mathcal{G}$ is not required to be a lattice.

In our model a *chronon*, that is, the non-decomposable time unit, corresponds to a time instant t belonging to the time domain $\mathbb{N}$. The granularity which establishes a biunivocal correspondence with the time domain is called *chronon granularity*, denoted as G_I. G_I is such that $G_I(i) = \{i\}$, for each $i \in \mathcal{IS}$. We assume that the chronon granularity G_I belongs to $\mathcal{G}$.

Example 1. Let *hours*, 3*hours*, *days*, 15*days*, *weeks*, *months*, 3*months*, *years*, 5*years*, and *decades* be granularities such that $hours \preceq 3hours \preceq days \preceq months \preceq 3months \preceq years \preceq 5years \preceq decades$, $days \preceq weeks$, and $days \preceq 15days$ with the usual meaning. The set of granularities $\mathcal{G}$ we assume in all the examples in the paper is {*hours*, 3*hours*, *days*, *months*, 3*months*, *years*, 5*years*, *decades*, G_I}. Note that this set is not a lattice. ◇

We now define the notion of *temporal interval*.

Definition 3. *(*Temporal Interval*). Let $G \in \mathcal{G}$ be a granularity and $i, j \in \mathcal{IS}$ be two indexes such that $i \leq j$. Then $[i, j]^G = \{G(k) \mid i \leq k \leq j, k \in \mathcal{IS}\}$ is called* temporal interval*, with respect to granularity G.* □

The temporal interval $[i, \infty]^G$ denotes an infinite countable set of granules, that is, $[i, \infty]^G = \{G(k) \mid k \geq i, k \in \mathcal{IS}\}$. The usual operations on sets (e.g., intersection, union, ...) are defined on temporal intervals.

3 Temporal Data Model

In this section we formalize the notion of temporal type, specifying its legal values, and introduce the subtype relation among temporal types. Then, we discuss temporal value conversions, introducing the notions of refinement and coercion functions. Finally, we introduce T_ODMG classes and objects, in which an attribute can be specified to be a temporal attribute.

3.1 Temporal Types and Values

In [5] we extended the ODMG type set with *temporal types*, handling temporal types and ODMG types in a uniform way. In this paper we propose a different definition of temporal types as parametric types. The notion of parametric or generic type is quite common in object-oriented languages, where it is used to define data and algorithms which structure does not depends on the types of data involved.

Let $\mathcal{ST}$ be the set of ODMG types, including class and literal types denoted by $\mathcal{OT}$ and $\mathcal{LT}$, respectively. We refer to these types as *static types*, to emphasize that they are non temporal types. A static type characterizes traditional properties, for which only the current value is stored in the database. For each temporal granularity $G \in \mathcal{G}$ and for each static (inner) type $\tau \in \mathcal{ST}$, a temporal type $temporal\langle G, \tau \rangle$ can be defined. Parameter G is bounded by the whole set $\mathcal{G}$ of granularities given in a database schema, whereas parameter τ ranges over object and literal ODMG types. Representing temporal types as parametric types, we stress the idea that each temporal type has the same structure and depends only and completely on its parameters, that is, its granularity and its inner (static) type. The set of temporal types, defined as follows, is denoted by $\mathcal{TT}$, and the set of T_ODMG types is denoted by $\mathcal{T} = \mathcal{ST} \bigcup \mathcal{TT}$.

Definition 4. *(*Temporal Types*). Let $\tau \in \mathcal{ST}$ be a static type and $G \in \mathcal{G}$ a granularity, $temporal\langle G, \tau \rangle$ is the* temporal type *corresponding to type τ and granularity G.* □

A temporal value of a temporal type $temporal\langle G, \tau \rangle$ is defined as a partial function that maps G-granules (referred to through their indexes) to τ values. We refer to the set of time instants for which this partial function is defined as the *domain* of the temporal value. In what follows, given a type $\tau \in \mathcal{T}$, the notation $[\![\tau]\!]$ denotes the set of legal values for type τ [5].

Example 2. Given a temporal attribute representing the population of a city, with type $temporal\langle days, int \rangle$, a legal value for such attribute is $v = \{\langle 06/11/2003, 15004\rangle, \langle 07/11/2003, 15032\rangle, \langle 08/11/2003, 15024\rangle\}_{days}$. v has been represented as a set of pairs of integer values, representing the daily value of the population, and granules of granularity *days*, that give the reference days. ◇

Let v be a value of type $temporal\langle G, \tau \rangle$, and $i \in \mathcal{IS}$ be an index, $v(i)$ denotes the value of v in the ith granule of G. Since temporal values are partial

functions, given a temporal value v, an index i may exist such that $v(i) = \perp$. Since we will usually denote granules through their labels, if v is a value of type $temporal\langle G, \tau\rangle$, and l^i_G is the label of the ith granule of G, $v(l^i_G)$ equivalently denotes $v(i)$, i.e, the value of v in the granule labeled by l^i_G.

Example 3. Given the temporal value of Example 2, suppose the index of granularity *days* associated with the label 06/11/2003 be 17, that is $l^{17}_{days} = 06/11/2003$, then the (non temporal) value corresponding to the label can be referenced as $v(06/11/2003) = 15004$. ◇

We now consider the subtype relation on temporal types. Since temporal types are parametric types, we first discuss the existing relations on parameters, i.e. granularities and static types. Among static types the usual notion of subtype can be devised. Specifically, for object types, the subtype relation relies on the inheritance relationships between interfaces and classes defined in a T_ODMG database schema. For the formal definition of the subtype relation on static types we refer to [5]. By contrast, the only relationship holding among granularities in the T_ODMG model is the finer than relationship, that is not suitable to model a subtype relation. Granularities in $\mathcal{G}$ have a common behaviour that can be easily modeled as an abstract data type, but no subtype relation can be devised among them. Then, the subtype relation between temporal types only relies on the existing subtype relation between inner static types.

Definition 5. *(*Temporal Subtypes*). Let τ_1, $\tau_2 \in \mathcal{TT}$ be temporal types, such that $\tau_1 = temporal\langle G, \tau_1'\rangle <: \tau_2 = temporal\langle G, \tau_2'\rangle$. Then τ_2 is a* subtype *of τ_1 (denoted as $\tau_2 <: \tau_1$) if and only if $\tau_2' <: \tau_1'$.* □

Subtyping has some important implications in object-oriented models. As expected, a temporal type that is subtype of another temporal type can *subsume* it. Information about the actual parameters of a temporal type, that is, the granularity and the inner type used to instantiate it in the database schema, are available at run time. Thus, the correctness of an assignment to a temporal attribute can be checked at run time. The same rule does not hold, in general, in object-oriented languages that treat parametric types performing what is known as *homogeneous translation* [9], that is, a translation to the original non parameterized language, that does not preserve type information about parameters at run time.

Proposition 1 *(*Subsumption*). Let τ_1, $\tau_2 \in \mathcal{TT}$ be temporal types, such that $\tau_1 = temporal\langle G, \tau_1'\rangle <: \tau_2 = temporal\langle G, \tau_2'\rangle$. A value v of type τ_2 can be used everywhere a value of type τ_1 is expected, that is, v has type τ_1.* ○

The second consequence of the subtype relationship is *extent inclusion*, that is, the property ensuring that the extent of a subtype is included in the extent of its supertype.

Proposition 2 *(*Extent Inclusion*). Let τ_1, $\tau_2 \in \mathcal{TT}$ be two temporal types such that $\tau_2 <: \tau_1$, then $[\![\, \tau_2 \,]\!] \subseteq [\![\, \tau_1 \,]\!]$.* ○

3.2 Conversion of Temporal Values

In our model, coercion and refinement functions allow the conversion of temporal values from one granularity to another. Coercion functions have been defined in [5] to convert temporal values to a coarser granularity in a meaningful way. Let $temporal\langle G, \tau\rangle$ and $temporal\langle H, \tau\rangle$ be two temporal types such that $G \prec H$. A *coercion function* $C : [\![temporal\langle G, \tau\rangle]\!] \rightarrow [\![temporal\langle H, \tau\rangle]\!]$ is a total function that maps values of type $temporal\langle G, \tau\rangle$ into values of type $temporal\langle H, \tau\rangle$. Coercion functions can be classified into three categories: *selective*, *aggregate*, and *user-defined* coercion functions. Selective coercion functions are `first`, `last`, `main`, `all`, and `proj(index)`. Coercion function `proj(index)`, for each granule in the coarser granularity, returns the value corresponding to the granule of position `index` of the finer one. Coercion function `first` and `last` are the obvious specialization of the previous one. Coercion function `main`, for each granule in the coarser granularity, returns the value which appears most frequently in the included granules of the finer one. Coercion function `all`, for each granule in the coarser granularity, returns the value which always appears in the included granules of the finer one if this value exists, the null value otherwise. Aggregate coercion functions are `min`, `max`, `avg`, and `sum` corresponding to the well-known SQL aggregate functions[2]. User-defined coercion functions correspond to methods declared in a class of the database schema.

Each coercion function introduced above actually corresponds to a family of coercion functions. That is, `sum` actually is a way to denote the set of functions $\{\texttt{sum}_{G\rightarrow H} \mid G, H \in \mathcal{G}, G \prec H\}$, where $\texttt{sum}_{G\rightarrow H} : [\![temporal\langle G, \tau\rangle]\!] \rightarrow [\![temporal\langle H, \tau\rangle]\!]$. Since, however, the behavior of each function in this set is the same, we will refer in what follows to a generic function `sum`, without explicitly specifying the granularities, if ambiguities do not arise. Some functions are meaningful only for some inner types. Specifically, `sum` and `avg` are meaningful only for numeric values, whereas `min` and `max` are meaningful only for values of ordered domains.

An important assumption made in [5] was that all attributes are *downward inheritable* [20]. This means that, for each pair of granularities G and H, such that $G \prec H$, and for each pair of indexes i, j, such that $G(i) \subseteq H(j)$, the value of v in granule i of G is the one in the jth granule of H. As we discussed in the introduction, the downward hereditary property is not always appropriate. If for instance the value of an attribute is stored with respect to months, then, depending on the attribute semantics, this may not imply that such value is the same for every day in the month. Thus, in this paper, we introduce *refinement functions*, that allow one to specify how to convert temporal values from a given granularity into values of a finer granularity in a meaningful way. Let $temporal\langle G, \tau\rangle$ and $temporal\langle H, \tau\rangle$ be two temporal types such that $G \prec H$. A *refinement function* $R : [\![temporal\langle H, \tau\rangle]\!] \rightarrow [\![temporal\langle G, \tau\rangle]\!]$ is a total function that maps values of type $temporal\langle H, \tau\rangle$ into values of type $temporal\langle G, \tau\rangle$. Specifically, we consider both predefined and user-defined refinement functions.

[2] In computing aggregate coercion functions we consider undefined values (i.e., values at granules i such that $v(i) = \perp$) as *null* values in OQL.

User-defined refinement functions correspond to methods declared in a class of the database schema. Predefined refinement functions include `restr` and `split`. Refinement function `restr` models those attributes for which downward inheritance is reasonable. This means that if a temporal property assumes value v on a granule g, value v also refers to any granule g' of the finer granularity included in g. Thus, given a value $v \in [\![temporal\ \langle H, \tau \rangle]\!]$, for each $j \in \mathcal{IS}$ $\texttt{restr}(v)(j) = v(i)$, where $i \in \mathcal{IS}$ is such that $G(j) \subseteq H(i)$. That is, the value of $\texttt{restr}(v)$ in granule j of G is the one in the ith granule of H. Refinement function `split`, by contrast, models those attributes for which downward inheritance is not adequate. The idea is to split the value assumed by the temporal value on each granule g of the coarser granularity among the granules g' of the finer granularity included in g. Thus, given a value $v \in [\![temporal \langle H, \tau \rangle]\!]$, for each $j \in \mathcal{IS}$ $\texttt{split}(v)(j) = v(i)/n$, where $i \in \mathcal{IS}$ is such that $G(j) \subseteq H(i)$ and $n = |\{h \mid h \in \mathcal{IS}, G(h) \subseteq H(i)\}|$. That is, the value of $\texttt{split}(v)$ is obtained by splitting the value of the ith granule of H, among the G granules included in $H(i)$. Each refinement function introduced above actually correspond to a family of functions, as discussed above for coercion functions. Moreover, refinement function `split` is meaningful only for numeric values.

Example 4. Given the temporal value v of type $temporal\langle months, int\rangle$ $\{\langle 01/2002, 372\rangle, \langle 02/2002, 420\rangle\}_{months}$, refinement function $\texttt{split}_{months \to days}$ applied to v results in the temporal value $\{\langle 01/01/2002, 12\rangle, \ldots, \langle 31/01/2002, 12\rangle, \langle 01/02/2002, 15\rangle, \ldots, \langle 28/02/2002, 15\rangle\}_{days}$. By contrast, refinement function $\texttt{restr}_{months \to days}$ applied to v results in the temporal value $\{\langle 01/01/2002, 372\rangle, \ldots, \langle 31/01/2002, 372\rangle, \langle 01/02/2002, 420\rangle, \ldots, \langle 28/02/2002, 420\rangle\}_{days}$. ◇

Since coercion and refinement functions are employed to convert temporal values from one granularity to another, two relevant properties can be devised for such functions. The first property refers to the *compositionality* of such functions, corresponding to the intuition that if we have three granularities G, H, I such that $G \prec H \prec I$ and a function `f`, the result of converting through `f` from G to H, and then from H to I, is the same of converting through `f` from G to I (and similarly for refinement functions).

Definition 6. (Compositionality). *Let* `f` *be a coercion/refinement function,* $G, H, I \in \mathcal{G}$ *be granularities such that* $G \prec H \prec I$ *(resp.,* $I \prec H \prec G$*),* `f` *is* compositional *if* $\forall \tau \in \mathcal{T}$, $\forall v \in [\![temporal\langle G, \tau\rangle]\!]$, $\texttt{f}_{H \to I}(\texttt{f}_{G \to H}(v)) = \texttt{f}_{G \to I}(v)$. □

Specifically, coercion functions `first`, `last`, `all`, `min`, `max`, and `sum` are compositional, whereas `proj`, `main`, `avg` are not. Rrefinement function `restr` is compositional, whereas `split` is not. Note that some functions (e.g., `split`, `avg`) would have been compositional if a stronger relationship, such as the *periodically groups into* [6], hold among granularities.

Example 5. Consider value v of type $temporal\langle days, int\rangle$ $\{\langle 08/02/2002, 15\rangle, \langle 01/11/2002, 6\rangle, \langle 4/11/2002, 6\rangle, \langle 25/11/2002, 6\rangle, \langle 28/12/2002, 15\rangle\}_{days}$. If we first apply function $\texttt{sum}_{days \to months}$, and then function $\texttt{sum}_{months \to years}$, the

temporal value $\{\langle 2002, 48\rangle\}_{years}$ is obtained, as if we directly apply function $\texttt{sum}_{days\rightarrow years}$ to v. By contrast, since `main` is not compositional, if we first apply function $\texttt{main}_{days\rightarrow months}$, and then function $\texttt{main}_{months\rightarrow years}$, we obtain the temporal value $\{\langle 2002, 15\rangle\}_{years}$, while if we directly apply function $\texttt{main}_{days\rightarrow years}$ we obtain the temporal value $\{\langle 2002, 6\rangle\}_{years}$. ◇

The second property refers to the *invertibility* of such functions, corresponding to the intuition that if we have two granularities and a pair of functions `f` and `g`, the effect of converting a temporal value v through `f`, and then converting back the result through `g`, returns the original value v. When converting from a finer to a coarser granularity, we loose details on the temporal value, and we cannot expect to be able to re-obtain them if we convert back to the finer granularity. By contrast, when converting from a coarser to a finer granularity, we introduce some details that we should be able to forget, if we are no more interested in them, re-obtaining the original value. The first aspect is captured by the notion of *quasi-inverse* functions, that takes into account that some details are lost and thus some imprecision is introduced. The second aspect is captured by the notion of *inverse* functions. There is an analogy with what happens in the object-oriented context where if we cast up an object to a superclass, and then recast it down to its original class, we are not able to re-obtain the details we have forgot with the cast up, whereas if we cast down an object to a subclass, we are then able to re-obtain the original object if it is recast up to its original class.

Definition 7. (Inverse). *Let* `f` *be a coercion function,* `g` *be a refinement function,* $G, H \in \mathcal{G}$ *be granularities such that* $G \prec H$, `f` *and* `g` *are* inverse *if* $\forall \tau \in \mathcal{T}$, $\forall v \in [\![temporal\langle H, \tau\rangle]\!]$, $\texttt{f}_{G\rightarrow H}(\texttt{g}_{H\rightarrow G}(v)) = v$. □

Specifically, coercion function `sum` is the inverse of refinement function `split`, whereas refinement function `restr` is the inverse of coercion functions `first`, `last`, `min`, `max`, `proj`, `main`, `all`, `avg`.

Definition 8. (Quasi-Inverse). *Let* `f` *be a coercion function,* `g` *be a refinement function,* $G, H \in \mathcal{G}$ *be granularities such that* $G \prec H$, Δ *be a quantification of the maximum allowable error,* `f` *and* `g` *are* quasi-inverse *if* $\forall \tau \in \mathcal{T}$, $\forall v \in [\![temporal\langle G, \tau\rangle]\!]$, $\forall i \in \mathcal{IS}$, $\texttt{g}_{H\rightarrow G}(\texttt{f}_{G\rightarrow H}(v))(i) \in v(i) \pm \Delta$. □

While the notion of inverse is meaningful for all the attribute domains, the notion of quasi-inverse is only meaningful for numeric attributes. Whether two functions are quasi-inverse depends on how Δ is set. A typical setting of Δ could be, for each H-granule j, $(max_j - min_j)/n_j$ where $max_j = max\{v(h) \mid h \in \mathcal{IS}, G(h) \subseteq H(j)\}$, $min_j = max\{v(h) \mid h \in \mathcal{IS}, G(h) \subseteq H(j)\}$, $n_j = |\{h \mid h \in \mathcal{IS}, G(h) \subseteq H(j)\}|$. The global Δ could then be determined as the maximum over the Δ_j's determined in this way. With such a setting of Δ, for instance, coercion function `sum` is the quasi-inverse of refinement function `split`, whereas refinement function `restr` is the quasi-inverse of coercion function `avg`.

Example 6. Consider the temporal value $v_0 = \{\langle 2001, 36\rangle\}_{years}$, the refinement function $\texttt{split}_{years\rightarrow months}$ and the coercion function $\texttt{sum}_{months\rightarrow years}$. Since

split and *sum* are inverse, $\texttt{split}_{years \to months}(v_0)$ is the temporal value $\{\langle 01/2001, 3\rangle, \langle 02/2001, 3\rangle, \ldots, \langle 12/2001, 3\rangle\}_{months} = v'_0$, and $\texttt{sum}_{months \to years}(v'_0) = v_0$. Consider now $\Delta = 5$ and the temporal value $v = \{\langle 01/2002, 3\rangle, \langle 02/2002, 2\rangle, \langle 03/2002, 4\rangle, \langle 04/2002, 5\rangle, \langle 05/2002, 1\rangle, \langle 06/2002, 2\rangle, \langle 07/2002, 4\rangle, \langle 08/2002, 3\rangle, \langle 09/2002, 3\rangle, \langle 10/2002, 6\rangle, \langle 11/2002, 0\rangle, \langle 12/2002, 3\rangle\}_{months}$. Since *avg* and *restr* are quasi-inverse, $\texttt{avg}_{months \to years}(v)$ is the temporal value $\{\langle 2002, 3\rangle\}_{years} = v'$, and $\texttt{restr}_{years \to months}(v')$ is the temporal value associating 3 with each month of 2002, and, for each month of 2002, the difference between the value in v and 3 is less than Δ. ◇

3.3 Classes and Objects

A class declaration consists of a class identifier, that represents the object type of the class, and a set of attributes[3]. Each attribute has a name and a type. An attribute of a T_ODMG class can be temporal, if we are interested in storing values it has taken over time, or static, if only the current value of the attribute is kept. Temporal attributes have a temporal type at a certain granularity as domain, whereas static attributes have a static type as domain. A T_ODMG class specification is a pair $(i, attr)$, where $i \in \mathcal{OT}$ is the type identifier, and *attr* is an attribute specification, defined as follows.

Definition 9. (Attribute Specification without Attribute Refinement). *An* attribute specification *in a T_ODMG class is a set containing an element for each attribute of the class. Each element is a pair (a_{type}, a_{name}), where $a_{type} \in \mathcal{T}$ is the attribute domain, and a_{name} is the attribute name.* □

Example 7. An example of T_ODMG class specification is reported in Fig. 1, in which a class `city` has been specified with a static attribute `name` and a set of temporal attributes. The temporal attribute `mayor` stores the city mayor, that can be reelected every 5 years, as instance of class `politician`. Temporal attribute `population` records the number of inhabitants of the city, at the *days* granularity, and temporal attributes `temperature`, `rainfall` and `snowfall` give some climatic characteristics of the city, at granularity *months*. ◇

A T_ODMG object is a 5-tuple $(id, N, v, c, [i, j]^{G_I})$ where id is the object identifier, N is the set of object names, v is the object state, formally defined in the following definition, c is the most specific class to which the object belongs, and $[i, j]^{G_I}$ is a temporal interval representing the object lifespan, that is, the interval during which the object exists, expressed at the chronon granularity.

Definition 10. (Object State). *Given an object o, its* state v *is a tuple* $(a_1 : v_1, \ldots, a_n : v_n)$*, where each v_i, $1 \leq i \leq n$, is a static or temporal value.* □

[3] We do not include methods and relationships in class specification because the focus of the paper is attribute refinement. For the same reason, we disregard interfaces, since they only allow to specify methods.

```
class city{
 attribute string name;
 attribute temporal⟨5years,politician⟩ mayor;
 attribute temporal⟨days,int⟩ population;
 attribute temporal⟨months,float⟩ temperature;
 attribute temporal⟨months,int⟩ rainfall;
 attribute temporal⟨months,int⟩ snowfall; };
```

Fig. 1. Example of T_ODMG class specification * sistemare layout table and attribute * use of code environment

```
name={``Milan"};
major={⟨2000-2005, o2⟩,⟨2005-2009, o3⟩}5years;
population={⟨1/3/2003, 800⟩,⟨2/3/2003, 830⟩,...}days;
temperature={⟨1/2003, 10, 5⟩,⟨2/20038, 9⟩,...}months;
rainfall={⟨1/2003, 25⟩,⟨2/2003, 21⟩,...}months;
snowfall={⟨1/2003, 5⟩,⟨2/2003, 7⟩,...}months;
```

Fig. 2. Example of T_ODMG object state

Example 8. Consider class `city` of Example 7. Let $\mathtt{o}_1$ be an object identifier, o_2 and o_3 be object identifiers of type `politician`, and `milan` be an object name. An object of class `city` is $(\mathtt{o}_1, \{\mathtt{milan}\}, v, \mathtt{city}, [1, \infty]^{G_I})$, where v is the object state depicted in Fig. 2. ◇

4 Attribute Refinement

In this section we discuss how to ensure substitutability for temporal attribute refinement, establishing the consistency conditions for a T_ODMG database schema. We focus on those attribute refinements that can be devised as *safe* at schema definition time. Then, we give the specification of T_ODMG classes that include refinements of temporal attributes. Finally, we analyze in detail the different cases of safe refinement of temporal attributes.

4.1 Safe Refinement of Temporal Attribute

In T_ODMG [5] inherited features can be redefined performing *covariant* redefinitions [1]. The adopted approach considers property domains as integrity constraints to be checked at run time, rather than dealing with them as type constraints. Additional run time checks are needed, as widely demonstrated in the object-oriented literature, because attribute redefinition is an unsafe operation. Unsafety also arises for covariant refinement of temporal attributes: if we

override a temporal attribute defined with type $\tau_1 = temporal\langle G, \tau_1' \rangle$ with a type $\tau_2 = temporal\langle H, \tau_2' \rangle$ such that $\tau_2 <: \tau_1$, such redefinition requires the introduction of additional run time checks to prevent run time errors. By contrast, for temporal attributes we are able to devise a set of significant refinements that are *safe*, i.e., for which we are able, at schema definition time, to state that no run time errors can arise.

When specializing a temporal attribute domain a change of granularity can be a more realistic need than a change (even than a specialization) of the inner type, involving a different domain for the attribute. The refinement of the granularity of a temporal attribute, indeed, actually represents a change of the level of detail for the attribute. Depending on the application domain and on the attribute semantics, in the subclass, for instance, one could be interested in keeping the attribute values at a greater level of detail, or in decreasing the level of detail at which an attribute value is recorded. Since such temporal attribute refinements involve only the granularities, their consistency can be checked at schema definition time. Specifically, refinements that refine the domain of a temporal attribute with a finer granulary, or with a coarser one, are safe. The only exception is represented by refinements that involve *gap granularities* [6], that have non contiguous granules, and require additional run time checks to distinguish accesses to value related to non existing portions of granules (throwing run time exceptions), from accesses to undefined temporal values (that results in null values). For instance, an additional check is needed to distinguish if an attribute value, stored at granularity *bweeks*, is accessed on a Sunday (which does not make sense) from the case in which the attribute value is accessed on a working day for which no value has been stored.

A temporal database schema with safe attribute refinements, in order to be defined as consistent, should satisfy some properties related to object casts. First, *substitutability* must be satisfied, according to which an object with dynamic type c, can be used as having any supertype of c. Second, a *cast down* to a class c' of an object with static type c and dynamic type c', such that $c' <: c$, should not produce run time errors. Such a cast down would allow temporal refined attributes of class c' to be accessed as well. We want to enforce both properties along an inheritance hierarchy involving an arbitrary number of classes in which temporal attributes are refined in a safe way, that is, by refining their granularities.

Substitutability requires to convert attribute values to different domains. For the safe refinements of temporal attributes we discussed, coercion and refinement functions defined in Section 3 and referred in what follow as *conversion* functions, are employed. Specifically, when refining an attribute of type $temporal\langle G, \tau \rangle$ to a type $temporal\langle H, \tau \rangle$, a pair of conversion functions must be included in the specification of the attribute. The first function will be employed for accessing the attribute value at granularity G, that is, it converts the attribute value from type $temporal\langle H, \tau \rangle$ to type $temporal\langle G, \tau \rangle$. The second conversion function will be used for updating the attribute value at granularity G, that is, it performs the inverse conversion. In particular, if $H \prec G$, for the attribute, a coercion function

for accessing the attribute value and a refinement function for updating it should be provided. By contrast, if $G \prec H$, the specification should include a refinement function for accessing the value and a coercion function for updating it. The following definition formalizes the notion of safe refinement.

Definition 11. (Safe Refinement of Temporal Attributes). *Given an attribute a defined in a class c' with type $\tau_1 = temporal\langle G, \tau\rangle$ and refined in a class c, subclass of c', with type $\tau_2 = temporal\langle H, \tau\rangle$, such refinement is* safe *if one of the following conditions holds:*

- *$H \prec G$, and the pair $\langle af, uf\rangle$ is specified for attribute a in class c, where af is a coercion function and uf is a refinement function;*
- *$G \prec H$, and the pair $\langle af, uf\rangle$ is specified for attribute a in class c, where af is a refinement function and uf is a coercion function.* □

Whenever a class specification includes a pair of conversion functions for each temporal attribute refinement specified for the class, such functions are used to access and to update the attributes. Then, cast semantics must be modified to take into account the presence of such refinements. Before formalizing cast semantics, we formalize access and update operations, since we consider cast as an operation required in order to access and to update temporal attributes.[4] Specifically, accesses we focus on require the value of an object attribute in a specified granule. This form of access can be easily generalized to the access to the value of an object attribute in a temporal interval, that denotes a temporal value at the specified granularity, as those considered in [4]. Since this extension does not introduce new issues, and it complicates the notation used, we prefer to focus on single granule object accesses. The simplest forms of access and update do not require to perform any conversion of attribute temporal values. The only requirement is that an attribute with the specified name exists for the object that is accessed or updated. The following definition formalizes such simple operations.

Definition 12. (Simple Access and Update). *Given a class c, and a temporal attribute a defined, inherited, or refined in class c with type $temporal\langle G, \tau\rangle$, let o be an object of class c and v the value of attribute a in object o. Let finally l^i_G be a granule label, then $o.a \downarrow l^i_G$ denotes the* access *to the value of attribute a of the object denoted by o in granule l^i_G, that is, $v(l^i_G)$.*

Let $\overline{v}_G$ be a temporal value of type $temporal\langle G, \tau\rangle$. Then, the expression $u(o.a, \overline{v}_G)$ results in the update *of the value of attribute a with temporal value $\overline{v}_G$. If some of the values in $\overline{v}_G$ refer granules for which some values were already defined for a, the values in $\overline{v}_G$ are taken.* □

Suppose attribute a, first declared in class c', a superclass of c, with granularity H, is refined in class c with granularity G. For substitutability we can access and

[4] Since the focus of the paper is on refinement of temporal attributes, we consider only accesses and updates to such kind of attributes, disregarding attributes whose domain is a static type.

update attribute a as it is defined in class c'. This requires casting o to type c' and specifying a granule of granularity H for the access, or a temporal value at granularity H for the update. The following definition formalizes the semantics of access and update to a temporal attribute requiring a cast to a superclass, specifying how the cast is mapped into conversions of temporal values.

Definition 13. (Access and Update with Cast Up). *Given two classes, c and c', with c subclass of c', and a temporal attribute a defined in c' with type $temporal\langle H, \tau\rangle$ and refined in c for the first time in the inheritance hierarchy with type $temporal\langle G, \tau\rangle$, such that $G \prec H$ or $H \prec G$. Suppose the refinement or coercion function specified for the attribute access be $af : [\![temporal\langle G, \tau\rangle]\!] \rightarrow [\![temporal\langle H, \tau\rangle]\!]$. Let o be an object of class c and v the temporal value of attribute a in object o. Finally, let l^i_H be a granule label of granularity H. Then,*

$$(c')o.a \downarrow l^i_H = af_{G\rightarrow H}(v)(l^i_G).$$

Let the refinement or coercion function specified for the attribute update be $uf : [\![temporal\langle H, \tau\rangle]\!] \rightarrow [\![temporal\langle G, \tau\rangle]\!]$, and let $\overline{v}_H$ be a temporal value of type $temporal\langle H, \tau\rangle$. Then,

$$u((c')o.a, \overline{v}_H) = v \cup uf_{H\rightarrow G}(\overline{v}_H).$$ □

Example 9. Consider class c' and its direct subclass c, such that attribute a is refined from $temporal\langle years, integer\rangle$ in c' to $temporal\langle months, integer\rangle$ in c, with coercion function *sum* specified for access. Let o be an object instance of class c whose value for attribute a is value v of Example 6. Then $o.a \downarrow 03/2002 = 4$ and $(c')o.a \downarrow 2002 = sum_{months\rightarrow years}(v)(2002) = 36$. Given *split* as the refinement function for update, the update $u(o.a, \{\langle 01/2002, 10\rangle, \langle 02/2002, 12\rangle\})$ followed by the update $u(o.a, \{\langle 2003, 60\rangle\})$ results in value v=$\{\langle 01/2002, 10\rangle$, $\langle 02/2002, 12\rangle$,. . .,$\langle 12/2002, 3\rangle$,$\langle 01/2003, 5\rangle$, $\langle 02/2003, 5\rangle$,$\langle 03/2003, 5\rangle$,$\langle 04/2003, 5\rangle$, $\langle 05/2003, 5\rangle$,$\langle 06/2003, 5\rangle$, $\langle 07/2003, 5\rangle$,$\langle 08/2003, 5\rangle$,$\langle 09/2003, 5\rangle$,$\langle 10/2003, 5\rangle$, $\langle 11/2003, 5\rangle$,$\langle 12/2003, 5\rangle\}_{months}$. ◇

If more than one refinement for temporal attribute a is specified along the inheritance hierarchy, the semantics just formalized is extended in a straightforward way. We simply move up through the inheritance hierarchy that involves refinement for the temporal attribute a, mapping object cast with a step by step conversion of the temporal value to access or used for the update. At each step, the corresponding access or update conversion function is applied, obtaining the temporal value that will be converted at the following step. If the conversion functions applied are compositional (cf. Definition 6), the value to be returned can be obtained by applying the conversion function only once. If they are not compositional, by contrast, a sequence of conversions must be performed.

When along the inheritance hierarchy a chain of refinements involves, for the same temporal attribute, more than once the same granularity, the refinement is said *circular*. Such kind of refinements require the conversion functions specified for the refinement be inverse (cf. Definition 7) or quasi-inverse (cf. Definition 8), to ensure that the error introduced by the conversion process does not exceed an established bound. Circular refinements are discussed in Section 4.3.

The semantics of cast down is simpler than that of cast up, because, as for simple access, it does not involve value conversions. To perform a cast down of an object, in order to access or to update one of its temporal attributes, we must ensure that the object dynamic type be the object type specified for the cast. If it is, then we simply perform the access or the update with a granule or a temporal value of the right granularity. The following definition formalizes the notion of access and update involving a cast down.

Definition 14. (ccess and Update with Cast Down). *Given two classes, c and c', with c subclass of c', and a temporal attribute a defined in c' and refined in c with type $temporal\langle G, \tau\rangle$, let o be an object with dynamic type c and static type c', v the temporal value of attribute a in object o, l^i_G a granule label. Then, the object access $(c)o.a \downarrow l^i_G$ evaluates to $v(l^i_G)$. Finally, let $\overline{v}_G$ a legal value for attribute a at granularity G. Then, the object update $u((c)o.a, \overline{v}_G)$ evaluates to $v' = v \cup \overline{v}_G$.* □

4.2 Classes with Refined Attributes

T_ODMG class specification has to be extended to handle refined attributes. Specifically, attribute specification (cf. Definition 9) must be modified, by introducing conversion functions to ensure substitutability. Then, if an attribute specification is a refinement of an inherited attribute, it also contains a pair of conversion functions, one for accessing the attribute value and one for updating it using a granularity specified in a superclass definition. The following definition formalizes attribute specification with refinement of temporal attributes.

Definition 15. (Attribute Specification). *A T_ODMG*attribute specification *attr is a set containing an element for each attribute of the class. Each element is a 3-tuple $(a_{type}, a_{name}, a_{ref})$, where $a_{type} \in \mathcal{T}$ is the attribute domain; a_{name} is the attribute name; and $a_{ref} = \langle af, uf\rangle$ are the conversion functions, af for access and uf for update.* □

Example 10. Fig. 3 extends the database schema defined in Fig. 1, by declaring class `mainCity`, subclass of class `city`, and class `mountainCity`, subclass of `mainCity`. Class `mainCity` models main cities in a country, for which we maintain also information about pollution related to CO2 concentration, stored in attribute `co2`. Some of the attributes defined in class `city` have been refined in class `mainCity`, that requires a detailed meteorological monitoring of factors that influence pollution. Thus, attributes `temperature` and `rainfall` have been refined with finer granularities. For the same reason, attributes `windSpeed` and `windDir`, that model wind speed and direction, have been added to the class. Finally, attributes `snowfall` and `population` have been refined with coarser granularities. Attributes `name` and `mayor` are simply inherited.

Class `mountainCity` models ski touristic locations. Such locations require detailed information about snow conditions, thus attributes `snowfall` and `temperature` have been refined with finer granularities and attribute `snowheight` has

```
class mainCity extends city{
 ref attribute temporal⟨months, int⟩ population ⟨restr, last⟩;
 ref attribute temporal⟨3hours, float⟩ temperature⟨max, restr⟩;
 ref attribute temporal⟨days, int⟩ rainfall ⟨sum, split⟩;
 ref attribute temporal⟨years, int⟩ snowfall ⟨split, sum⟩;
 attribute temporal⟨3hours,string⟩ windDir;
 attribute temporal⟨3hours,float⟩ windSpeed;
 attribute temporal⟨3hours,float⟩ co2; };

class mountainCity extends mainCity{
 ref attribute temporal⟨decades, politician⟩ mayor ⟨restr, last⟩;
 ref attribute temporal⟨3months, int⟩ population ⟨restr, avg⟩;
 ref attribute temporal⟨hours, float⟩ temperature ⟨min, restr⟩;
 ref attribute temporal⟨years, int⟩ rainfall ⟨split, sum⟩;
 ref attribute temporal⟨days, int⟩ snowfall ⟨sum, split⟩;
 ref attribute temporal⟨days,string⟩ windDir ⟨restr, main⟩;
 ref attribute temporal⟨days,float⟩ windSpeed ⟨restr, max⟩;
 ref attribute temporal⟨3months,float⟩ co2 ⟨restr, avg⟩;
 attribute temporal⟨days,int⟩ snowheight; };
```

Fig. 3. Example of T_ODMG class specifications with refined attributes

been added to the class. Moreover, information about wind speed and direction are now stored with granularity *days*, while attributes `population` and `co2` have been refined with coarser granularity $3months$, modeling seasonal variations on such values. Finally, attribute `mayor` have been refined with granularity *decades*. Adequate pairs of conversion functions are specified for refined attributes in both class declarations, then the refinements are safe according to Definition 11. ◇

4.3 Analysis of Safe Refinements of Temporal Attributes

In this section we analyze different cases of safe refinement of temporal attributes that can arise in a T_ODMG database schema, referring the schema given in Fig. 1 and in Fig. 3 and presented in Example 10.

Refinement to coarser granularities (↑) Attribute `population` is defined in class `city` with granularity *days*, then refined in class `mainCity` with granularity *months* and in class `mountainCity` with granularity $3months$. Each attribute refinement specifies a pair of conversion functions. Functions declared for accessing the value belong to the same family *restr*, that is compositional, then, an access to attribute `population` with temporal value v of an object o with dynamic type `mountainCity`, with respect to class `city`, denoted by $(city)o.population \downarrow l^i_{days}$, results in the value $restr_{3months \to days}(v)(l^i_{days})$. By contrast, the update functions belong to families *last* and *avg*, then an update of `population` value with the temporal value $\overline{v}_{days}$, of type $temporal\langle days, int\rangle$,

denoted by $u((city)o.population, \overline{v}_{days})$, is performed by applying the right sequence of coercion functions on such value. Then, the update results in $v \cup avg_{months \rightarrow 3months}(last_{days \rightarrow months}(\overline{v}_{days}))$.

Refinement to finer granularities ($\downarrow$) Attribute `temperature` is defined in class `city` with granularity *months*, then refined in class `mainCity` with granularity $3hours$ and in class `mountainCity` with granularity *hours*. Functions declared for accessing the attribute value are *max* and *min*. Belonging to different families, they must be applied in the right sequence when performing an access involving a cast up. Then, $(city)o.population \downarrow l^i_{months}$, is equivalent to $max_{3hours \rightarrow months}(min_{hours \rightarrow 3hours}(v))(l^i_{months})$, with l^i_{months} granule of granularity *months* and v value of attribute `temperature`. Functions declared for updating the value belong to the same family *restr*, that is compositional, then, an update to attribute `temperature` with temporal value v of an object o with dynamic type `mountainCity`, with respect to class `city`, denoted by $u((city)o.population, \overline{v}_{months})$, can be performed by directly converting $restr_{months \rightarrow days}(\overline{v}_{months})$ and by appending such value to the existing temporal attribute value.

Refinement to a coarser and then to a finer granularity ($\wedge$) Attribute `snowfall` is defined in class `city` with granularity *months*, then refined in class `mainCity` with granularity *years* and in class `mountainCity` with granularity *days*. Then, attribute specification in class `maincity` requires the declaration of a refinement function for accessing the value and of a coercion function for updating it, and such functions are *sum* and *split*, whereas the attribute specification in class `mountainCity` requires the declaration of a coercion function for accessing the attribute value and of a refinement function for updating it, and such functions are *sum* and *split*, respectively. The access to attribute `snowfall` with temporal value v of an object o with dynamic type `mountainCity` with respect to class `city`, denoted by $(city)o.snowfall \downarrow l^i_{months}$, is equivalent to $split_{years \rightarrow months}(sum_{days \rightarrow years}(v))(l^i_{months})$. In a similar way, for executing the update $u((city)o.snowfall, \overline{v}_{months})$, the value $\overline{v}_{months}$ is converted by applying the conversion $split_{years \rightarrow days}(sum_{months \rightarrow years}(\overline{v}_{months}))$ and by appending such value to the existing temporal attribute value.

Circular refinement through a coarser granularity ($\triangle$) Suppose attribute `snowfall` is refined in class `mountainCity` with granularity *months*, instead of granularity *days*. This is a particular case of the preceding one, that models a class hierarchy in which an attribute is first refined with a coarser granularity and then re-refined to the initial granularity of the hierarchy. Since the selected conversion function for performing both access and update belong to families that are devised to be quasi-inverse with respect to Definition 8, we can ensure that the conversions performed do not introduce an error greater than the specified bound Δ.

Refinement to a finer and then to a coarser granularity ($\vee$) Attribute `rainfall` is defined in class `city` with granularity *months*, then refined in class `mainCity` with granularity *days* and in class `mountainCity` with granularity *years*. Then, attribute specification in class `maincity` requires the declaration

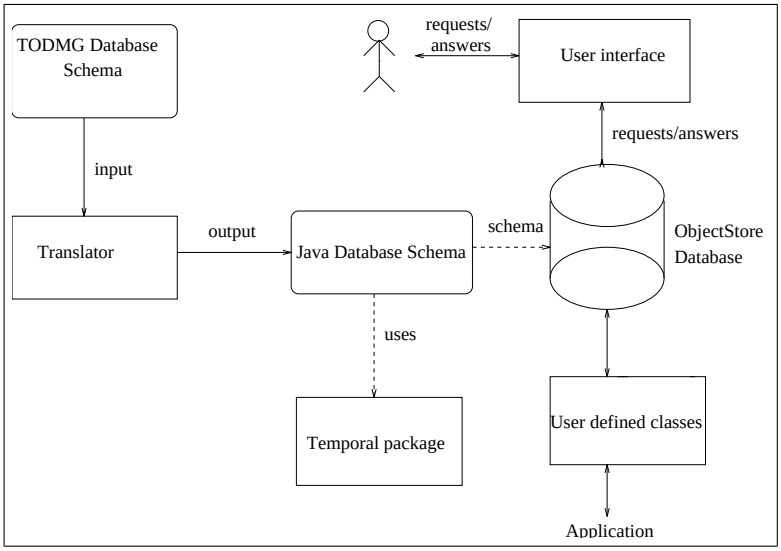

Fig. 4. System architecture

of a refinement function for updating the value and of a coercion function for accessing it, and such functions are *split* and *sum*; whereas the specification of `rainfall` in class `mountainCity` requires the declaration of a coercion function for accessing the attribute value and of a refinement function for updating it, and such functions are *sum* and *split*, respectively. Such attribute refinement is the logical inverse of that specified for attribute `snowfall` discussed above. The access to attribute `rainfall` with temporal value v of an object o with dynamic type `mountainCity` with respect to class `city`, denoted by $(city)o.rainfall \downarrow l^i_{months}$, is equivalent to $sum_{days \to months}(split_{years \to days}(v))(l^i_{months})$. For executing the update $u((city)o.rainfall, \overline{v}_{months})$, the value $\overline{v}_{months}$ must be converted to value $sum_{days \to years}(split_{months \to days}(\overline{v}_{months}))$, and then appended to the existing temporal attribute value v.

Circular refinement through a finer granularity (∇) Suppose attribute `rainfall` is refined in class `mountainCity` with granularity *months*, instead of granularity *years*. This is a particular case of the preceding one, that models a class hierarchy in which an attribute is first refined with a finer granularity and then re-refined to the initial granularity of the hierarchy. Since the selected conversion function for performing both access and update belong to families that are devised to be quasi-inverse with respect to Definition 8, we can ensure that the conversions performed do not introduce an error greater than the specified bound Δ.

5 Implementation Issues

The approach presented in this paper has been implemented in the T_ODMG prototype implementation realized in ObjectStore Java PSE Pro [17]. Note that ObjectStore Java does not support parametric types; however, no other Java-based OODBMS support them. The implementation is not based on ObjectStore specific features, thus it can be ported with little effort to other ODMG compliant Java based OODBMSs. Fig. 4 shows the overall architecture of the implementation. Main components of our prototype are: a *translator*, which takes as input a text file containing the definition of a T_ODMG database schema and returns the Java translation of the database schema; a `TemporalP` *package*, which is a Java library implementing multiple granularity temporal data; a *graphical user interface*, that supports the basic query and update operations on temporal objects and works on every set of Java classes obtained by the translation of a T_ODMG schema, thanks to the use of Java reflection. The Java classes obtained from the translator can also be directly used in user-defined classes of a user application. In the remainder of this section we briefly present the basic principles of temporal data handling in the prototype and then discuss the key issues in attribute refinement.

5.1 Temporal Data Handling

The representation of time is based on the class `java.util.Date`, provided by Java to represent time instants. Time instants are measured in milliseconds, with reference to January 1st, 1970 at 00:00:00.000. Milliseconds thus represent the finest granularity we can handle in our time domain, which is thus discrete. Temporal intervals are represented by the start and end instants of the interval, though the internal representation is a pair of `long` values. We have adopted such an approach because intervals are used as values of attributes of persistent objects and the class `java.util.Date` is not persistence capable [17]. The classes `OpenInterval` and `CloseInterval` implement open (i.e., of the form $[t, \infty]$) and closed (i.e., of the form $[t_1, t_2]$) temporal intervals, respectively. The classes provide operations on temporal intervals, such as intersection, test for inclusion, test for contiguity.

Temporal values are instances of the class `Temporal` of the `TemporalP` package. Each value of temporal type $temporal\langle G, \tau\rangle$ is handled as an object instance of class `Temporal`. Class `Temporal` has attributes: `type`, a string that stores the inner type τ; `granularity`, a string that stores the granularity G; `value`, an object instance of class `Temporal_List`, that stores the actual temporal value. In the current prototype, since we are not addressing efficiency requirements yet, the management of temporal values is very simple. A temporal value is simply represented as a double linked list. The list elements contain `value`, that is, an `Object` corresponding to the value, and `value_interval`, and `CloseInterval` corresponding to the granule to which such value refers. Note that, though according to the generic definition of granularity a granule corresponds to a generic

subset of the time domain, the various operations on temporal values can be implemented more efficiently if a granule corresponds to an interval on the time domain, that is, to a set of contiguous time instants. Since in most common granularity models, granules are intervals on the time domain, we adopt this more restricted notion in our implementation.[5] In addition, if a value is constant for some consecutive granules, then a single value is stored in the list, whose corresponding temporal interval is the union of the consecutive granules.

The implementation of temporal values as a sorted collection of interval-value pairs bring some similarity with the `HistoryOnAssociation` pattern [2]. However, since the history patterns considered by Anderson relate to transaction time, a single time instant is related to each value, corresponding to the instant in which the value is updated. Moreover, no multiple granularity management is considered in [2].

All temporal values are handled as instances of class `Temporal`. In this class both the attributes storing the granularity and the inner type of temporal attributes are of type `String`. Thus, Java reflective features are used to invoke methods of the class whose name is stored in the temporal attribute. The information on the temporal attribute inner type and granularity, though a schema information, is replicated for any object of the class. We have introduced such redundancy to improve performance.[6] The inner type is used by the mutator methods to ensure that the values associated with each granule are of the appropriate type and the granularity G is used by the mutator methods to ensure that each interval corresponds to a G granule or to a set of contiguous G granules.

For what concerns temporal granularities, since a granularity is a mapping, we do not need to create a new object for each granule, rather we can "implement" the mapping through methods. The index set $\mathcal{IS}$ has been implemented by the Java type `long` and granule labels are implemented by the Java type `String`. Three different information thus identify a granule of a granularity: its index; its textual representation, or label; the (closed) interval on the time domain it corresponds to. Thus, for each granularity two methods are defined: `indexToInterval`, which takes as input an index and returns the corresponding time interval on the time domain and `labelToIndex`, which takes as input the label of a granule and returns the corresponding index. In addition granularities store information related to the finer than relationship. Thus, for each granularity two attributes are defined: `finer`, storing the set of granularities finer than the considered one; `coarser`, storing the set of granularities coarser than the considered one.

[5] Note that this restriction still allows us to handle granularities that do not cover all the time instants in the time domain, such as the *business week* granularity.

[6] We could have associated these information only once with the attribute definition in the Java class corresponding to the translation of the class containing the attribute. However, under such an approach, this class definition would have been accessed through reflection each time the value of the attribute had needed to be updated for an object.

```
class Redefinition { private String g;
                     private String gSuper;
                     private String af;
                     private String uf;}

class CRFunction { public static Object sum(Object[] v)...;
                   public static Object avg(Object[] v)...;
                   public static Object min(Object[] v)...;
                   public static Object max(Object[] v)...;
                   public static Object restr(Object v)...;
                   public static Object split(Object v)...;
                   ...}
```

Fig. 5. Classes `Redefinition` and `CRFunction`

Temporal granularities are classes without instances, which correspond in Java to *abstract classes*. Each abstract class implementing a temporal granularity implements the `granularity` interface. In the current prototype, the predefined granularities are the ones of the Gregorian calendar, plus the *bweeks* granularity corresponding to business weeks. The approach we have taken to implement temporal granularities is however very flexible and allows users to add their own granularities, by simply defining a proper abstract class implementing the `granularity` interface described above. Upon definition of a new user-defined granularity the aciclicity of the graph corresponding to the finer than relationship is checked.

5.2 Attribute Refinement

Attribute refinement in T_ODMG has been implemented through two additional classes: `Redefinition.java` which implements the data structure storing the information related to attribute refinement; `CRFunction.java` which implements the predefined coercion and refinement functions. Class `Redefinition` contains four `String` attributes: `g`, which is the attribute current granularity; `gSuper`, which is the granularity of the attribute in the superclass; `af` and `uf` which are the associated function for access and for update, respectively. In addition, class `Redefinition` provides some methods which are used by the methods of the class in which the attribute has been refined to implement substitutability. Among them, the most relevant is method `convert` which converts the value of the attribute at granularity `g` into a value of granularity `gSuper`, that is, the one in the superclass. Such conversion is performed through the associated coercion/refinement function.

Class `CRFunction` implements the pre-defined coercion and refinement functions which have been presented in Section 3. Each function is a static method which takes as input an array of values (in case of coercion functions) or a

value (in case of refinement functions) and returns the converted value. For instance, suppose a temporal attribute domain is refined from granularity *years* to granularity *months* and the coercion function `sum` is specified. If we access such attribute for an object o in the superclass asking its value in year 1997, to coerce from granularity *months* to granularity *years*, method `CRFunction.sum` is invoked on the array of the twelve objects corresponding to the values taken by that attribute in the twelve months of 1997. These twelve values are appropriately extracted from the `value` attributes of the `Temporal_List` objects appearing in the `value` attribute of the `Temporal` object corresponding to the accessed temporal attribute of o.

6 Related Work

Issues related to temporal data models and query languages have been extensively investigated [15, 24], though most of the research and development efforts in the area of temporal databases have been carried out in the context of the relational model. However, several temporal object-oriented data models have been proposed in the literature [3, 13, 16, 18, 19, 22, 25]. In the relational context, some approaches [8, 12] have been developed that focus on handling and comparing temporal data at different granularities. Among the contributions included in [8], let us mention the concept of semantic assumption for temporal databases. A semantic assumption is a way for deriving implicit information from explicitly stored (relational) data. Coercion and refinement functions presented in our approach can be viewed as a specialization of semantic assumptions. In [12] multiple temporal granularity handling is also discussed. The focus is, however, on converting time instants (i.e., granules) from one granularity to another, rather than on converting temporal values from one granularity to another.

No comparable amount of work has been carried out to introduce temporal granularities in the context of the object-oriented data model. The introduction of multiple temporal granularities in an object-oriented data model poses additional issues with respect to the relational context, due to the semantic richness of such a model. Some object-oriented temporal data models deal with multiple temporal granularities [13, 18, 19, 22, 25]. Usually, the support of multiple temporal granularities in those models is provided as extension to the set of types of the temporal model. However, in most of these approaches the specification and management of different granularities, e.g., how to convert from a granularity to another, is completely left to the user. None of the cited proposals refers to the ODMG standard object model, which implies a strong dependence of each temporal model from the reference object model. Moreover, in contrast to the relational context, the introduction of temporal granularities in object-oriented data models is, in most cases, informal. For instance, none of the proposed approaches considers the impact of inheritance when multiple granularities are supported, nor they address issues concerning type refinement and substitutability.

We first presented a proposal for a temporal extension of the ODMG object-oriented data model in [3]. The current paper significantly extends [3] introduc-

ing, among other features, the support for multiple temporal granularities. A first approach towards the introduction of temporal granularities in an object-oriented model has been presented in [5]. This paper differs from and extends [5] in several respects. Specifically, a different notion of temporal types (as parametric types) is considered. Moreover, downward inheritance of attributes is not assumed, thus, the notion of refinement function is introduced to complement that of coercion function, and properties of such functions have been investigated. Finally, the comprehensive analysis of attribute refinement and its impact on substitutability is a novel contribution of this paper.

7 Conclusions

In this paper we have proposed a temporal multigranular object data model, in which temporal types are modeled as types parametric in two dimensions: the granularity and the inner (static) type. In the context of that model, we have then investigated attribute refinement along the inheritance hierarchy, allowing the granularity at which attribute values are stored to be modified in subclasses. A consequence of attribute refinement is that, to ensure substitutability, whenever an instance of a subclass is accessed as an instance of one of its superclasses, the need arises of converting the temporal value corresponding to an attribute value to the granularity of the attribute in the superclass. Conversions of temporal attributes from a granularity to another are realized through coercion and refinement functions, depending on attribute semantics, that are associated with attribute refinements in the database schema. To demonstrate the feasibility of the proposed approach, a Java prototype implementation of the data model has been developed on top of the ObjectStore PSE [17] ODMG compliant DBMS.

We are currently extending the work presented in the paper along different directions. A first extension of the data model concerns the support for *dynamic* attributes. A well-known problem of temporal databases is that the amount of stored data tends to increase very fast. Moreover, data acquired at a fine level of detail are useful when they are acquired but they often become less relevant after some time. In most cases, the level of detail at which data are needed depends on how recent data are. In [10] a multigranular temporal data model supporting the aggregation of different portions of the value of a temporal attribute at different levels of detail has been proposed. The problem of converting data from a granularity to another is extremely relevant also in that context.

Another relevant issue concerns data retrieval and query evaluation. Queries on temporal multigranular objects, such as path expressions traversing attributes stored at different granularities, require converting the traversed temporal value at the appropriate granularity. In the area of query languages, an interesting topic would be to associate a level of significance with the answers of queries. Indeed, due to the presence of coercion and refinement functions, we allow one to derive implicit information from explicitly stored one, that is, somehow, we allow flexible queries. Thus, an interesting issue would be to evaluate the query precision with respect to the stored information from a granularity point of

view. Finally, we are extending our prototype implementation in several respects. The extensions under development include the development of ad-hoc index structures well-suited both to associative and navigational access modalities to temporal objects. Moreover, we are interested in rewriting our prototype relying on the Java Data Objects [23] API, so that it can be used on top of any JDO implementation.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. F. Anderson. A Collection of History Patterns. In N. Harrison et al., editors, *Pattern Languages of Program Design, Vol. 4*, Addison-Wesley, 1999.
3. E. Bertino, E. Ferrari, G. Guerrini, and I. Merlo. Extending the ODMG Object Model with Time. In *Proc. 12th European Conference on Object-Oriented Programming*, LNCS 1445, pages 41–66, 1998.
4. E. Bertino, E. Ferrari, G. Guerrini, and I. Merlo. Navigating Through Multiple Temporal Granuarity Objects. In *IEEE Proc. 8th International Workshop on Temporal Representation and Reasoning*, pages 147–155, 2001.
5. E. Bertino, E. Ferrari, G. Guerrini, I. Merlo. T-ODMG: An ODMG Compliant Temporal Object Model Supporting Multiple Granularity Management. *Information Systems*, 28(8): 885-927, 2003.
6. C. Bettini, C.E. Dyreson, W.S. Evans, and R.T. Snodgrass. A Glossary of Time Granularity Concepts. In *Temporal Databases: Research and Practice*, LNCS 1399, pages 406–413, 1998.
7. C. Bettini, S. Jajodia, and X.S. Wang. *Time Granularities in Databases, Data Mining, and Temporal Reasoning*. Springer-Verlag, 2000.
8. C. Bettini, X. S. Wang, and S. Jajodia. Temporal Semantic Assumptions and Their Use in Databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):277–296, 1998.
9. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proc. of the ACM Conf. on Object-oriented Programming, Systems, Languages, and Applications*, pages 183–200, 1998.
10. E. Camossi, E. Bertino, G. Guerrini, and M. Mesiti. Evolution Specification of Multigranular Temporal Objects. In *IEEE Proc. 9th International Workshop on Temporal Representation and Reasoning*, pages 78–85, 2002.
11. R. Cattel et al. *The Object Database Standard: ODMG 3.0*. Morgan-Kaufmann, 1999.
12. C. E. Dyreson, W.S. Evans, H. Lin, and R. Snodgrass. Efficiently Supporting Temporal Granularities. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):568–587, 2000.
13. N. Edelweiss, J.P.M. de Oliveira, and B. Pernici. An Object-Oriented Temporal Model. In *Proc. Advanced Information Systems Engineering, CAiSE'93*, LNCS 685, pages 397–415, 1993.
14. C.S. Jensen and C.E. Dyreson. The Consensus Glossary of Temporal Database Concepts. In *Temporal Databases: Research and Practice*, LNCS 1399, pages 366–405, 1998.
15. C.S. Jensen and R. T. Snodgrass. Temporal Data Management. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):36–44, 1999.

16. W. Käfer and H. Schöning. Realizing a Temporal Complex-Object Data Model. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 266–275, 1992.
17. Object Design. ObjectStore Java API User Guide (ObjectStore 6.0). Available at `http://www.odi.com`, 1998.
18. M. T. Ozsu et al. TIGUKAT: A Uniform Behavioral Objectbase Management System. *VLDB Journal*, 4(3):445–492, 1995.
19. E. Rose and A. Segev. TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints. In *Proc. 10th Int'l Conf. on the Entity-Relationship Approach*, pages 205–229, 1991.
20. Y. Shoham. Temporal Logics in AI: Semantical and Ontological Considerations. *Artificial Intelligence*, 33(1):89–104, 1987.
21. R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publisher, 1995.
22. A. Steiner and M.C. Norrie. Implementing Temporal Databases in Object-Oriented Systems. In *Proc. of the 5th International Conference on Database Systems for Advanced Applications*, pages 381–390, 1997.
23. Sun Microsystems. Java Data Objects Version 1.0. Available at `http://access1.sun.com/jdo`, 2001.
24. Y. Wu, S. Jajodia, and X. S. Wang. Temporal Database Bibliography Update. In *Temporal Databases: Research and Practice*, LNCS 1399, pages 338–366, 1998.
25. G. Wuu and U. Dayal. A Uniform Model for Temporal and Versioned Object-Oriented Databases. In A. Tansel et al., editors, *Temporal Databases: Theory, Design, and Implementation*, pages 230–247. Benjamin/Cummings, 1993.