データベース・システム　79－5
プログラミング言語　26－5
（1990．　9．　7）

柔軟なアブストラクションを可能にする持続的オブジェクトのための一般モデルの構築にむけて

箕原辰夫　本田耕平　所真理雄
慶應義塾大学

従来のデータベースシステムとプログラミングシステムを統合した多形態なシステムを構築するために、プログラミングのためのモデルは、今までのシステムで開発されてきた様々な種類の概念を記述でき、それらを管理できるほど柔軟でなければならない。この論文は、今までそのような方向で提案されてきたシステムに、型と持続性の関連性という観点から、ある評価の基準を与える。この論文は、また、持続的なオブジェクトに基づいた、そのような多形態のシステムを構築するための、実体と連想関連に基づく一般的なモデルを提示する。

Towards Constructing A General Model For Persistent Objects Enabling Flexible Abstraction

Tatsuo Minohara　Kohei Honda　Mario Tokoro

Keio University

3-14-1 Hiyoshi, Kohoku-ku, Yokohama, 223 JAPAN

For constructing a polymorphous system which integrates the conventional database systems and the programming systems, the model for programming should be flexible so that it enables to describe various sorts of abstraction developed in the conventional systems and enables to manage them. This paper gives some criteria on such polymorphous systems which have been proposed, from the view point of the relationship between type and persistency. This paper also presents another general model founded on entities and associations for constructing such polymorphous systems based on persistent objects.

1. Introduction

Application programs which deal with persistent information face various problems because of impedance mismatches of semantics between the persistent information systems, i.e. the database management systems and the programming systems. The persistent information remains after the session of the execution of an application program. In such situations, the persistent information is shared by multiple applications concurrently or intermittently during a long period. We call the systems which integrate the conventional database management systems (or the information systems) with the programming languages systems *"persistent programming systems"*. The problems described bellow should be solved, when one wishes to achieve the integrated persistent programming systems. Particularly, for achieving the *open-ended* persistent programming systems, the systems should accept the applications which are programmed by using the conventional programming language systems. The following problems are inevitable in such open-ended systems.

Semantics Mismatches:

The applications described in different programming languages access a shared persistent information described in another different schema definition language. The semantics of types of the persistent programming systems, thus should be unified in the first place. However, the persistent information in a persistent programming system may be also accessed by applications written in the conventional programming languages. Consequently, the semantics of types of the persistent programming system should comprehend the semantics of types of various sorts of semantics of ordinary programming languages including type-less programming languages. In other words, the systems should be flexible enough to accept various sorts of abstraction. Interfaces which absorb the differences of the semantics of types between the applications and the persistent programming system are also required in the persistent programming system.

Evolutions:

The persistent information in persistent programming systems may be changed during their life time. The modifications may be applied on not only the value of persistent information, but also their schemes. The alterations of their schemes have to be announced to the applications which accesses the persistent information. Therefore, for enabling the alterations of the schemes of persistent information, the dynamic type mechanisms are indispensable. There are however the applications statically compiled by a compiler for some programming language system. These applications cannot be adaptive to the alteration. Although these applications should be recompiled basically, if the persistent programming system has the facilities of polymorphic types which let types have some latitude, the re-compilations may not be needed in some cases that the alterations of the schemes remain within the range of the latitude of the types. Although the versioning are also strong mechanisms for avoiding the re-compilations, the mechanisms cannot avoid the re-compilation in the cases that the propagations of the alterations of the persistent information to the applications are required. Furthermore, the methodologies of versioning for alterations of schemes have not been researched yet precisely. When the applications are recompiled, if the programs of these applications are written in a conventional programming language the source texts of the programs may have to be modified or have to be designed again in the worst cases. For minimizing such worst cases, the broad latitude of the types is required in the side of the persistent programming systems. Although dynamic typing mechanisms reduce such worst cases, the information of types of the persistent information could not be given to the programmers of applications who use the compiler which statically checks types of the programs, if the persistent programming systems adopt only dynamic typing mechanisms.

This paper focuses on persistent programming systems from the view point of type mechanisms and proposes a new model based on persistent objects for the systems. The type mechanisms are essential facilities for resolving the problems of the persistent programming systems argued above. Although the versioning mechanisms and the interface mechanisms are also indispensable mechanisms for achieving the persistent programming systems, we do not refer to the problems directly in this paper because of the limitation of the space. Particularly, assuming that the versioning mechanisms are prepared as basic facilities like the concurrency control or the recovery control mechanisms in the ordinary persistent

programming systems, we discuss the type mechanisms in the systems.

We try to describe the persistent programming systems that have been proposed in the field of the researches of the object-oriented database systems from the view point of the type mechanisms in Section 2. We classify these systems based on the criteria of the properties of persistency of objects and on the criteria of the binding mechanisms of types of persistent objects. We also discuss what functionalities are need to the persistent programming systems in addition to the facilities of the object-oriented database systems. We proposes a new persistent object model on the basis of the analyses of these object-oriented database systems in section 3. We conclude this paper with indicating the remaining problems and the future directions of our researches in section 4.

2. Types and Persistency

We try to give some interpretations on persistent programming systems, particularly on the object-oriented database systems [Atkinson 89]. In the object-oriented database systems, persistent information is stored in whole set of the *"persistent objects"*. Since an object is regarded as a primary component in the object-orientations, it is possible to design a unified persistent programming system by giving objects various sorts of characteristics. Namely, the design of the scheme of an object exerts an influence on whole schemes of the persistent programming system. Consequently, we also found on the object-orientations in order to achieve a unified system. The constructions of objects however are left more room to choice various sorts of design than the characteristics of an object. In the course of composing a persistent programming system, the designer must choice several important policies for the constructions of objects which dominate on whole design of the system. In the following subsections, we reviews these construction policies from the view point of the relations between types and persistency.

2.1. Types on Persistent Information

The persistent objects should be specified by some mechanism in the language of a persistent programming system. We describe how the persistent objects are specified in the language. For the specification of the persistent objects, three policies are classified as follows in our perspective:

Sets Based Persistency:

Set is one of primary constructors in information systems. The reason is considered as that information retrieval is equal to the extraction of appropriate elements from a set. The qualification of the extraction is given as a unified form of predicate which represents an intentional set corresponding to the extraction. All retrievals and information are integrated into the operations upon sets by adopting these predicates. In the conventional database systems, persistent information has been stored as the form of sets. For example, the relational database systems give persistency to the relations, which are sets that consists of tuples of values [Date 86] [Ullman 88].

Several persistent programming systems therefore throw the spotlight on sets as a type for persistency [Atkinson 87]. In these systems, the set is explicitly specified. GemStone [Maier 86], an object-oriented database systems based on Smalltalk-80 [Goldberg 83], adopts this policy. According to the descriptions of OPAL [Servio 89], the language of GemStone, the persistent information must be declared as a set type explicitly. The *"cluster"* corresponding to a persistent type should be specified in O++, the language of ODE (Object Database and Environment), for enabling to apply set operations on the objects which belong to the type [Agrawal 89]. In the languages of O_2 (e.g. CO_2, $LispO_2$) [Bancilhon 88], the *"with extension"* syntax can be used to a persistent type for indicating the objects of the type constructs a set as an extension of the type [Lecluse 89].

Types Based Persistency:

Although this policy is almost same as the previous policy, the set corresponding to a persistent type is implicitly specified. It is therefore possible to apply some set operations on the objects that belong to a persistent type automatically. In the conventional database systems, the relational model comes under this policy, since the description of a relation supposes the set of tuples in which the type of each tuple

follows the description. Daplex, one of earlier persistent programming languages for functional data models, assumes a set corresponding to each persistent type [Atkinson 87]. PS-Algol has been adopting the same approach [Atkinson 87]. ORION [Banerjee 87a] assumes the implicit set type corresponding to each type, and the implicit set types are automatically attached to the *"set"* type of the system as the subtypes of the *"set"* type.

Objects Based Persistency:

Each object can have the property of persistency individually in this policy. In previous two policies, the persistent objects are managed in accordance with the persistent types. In this policy, however, the type is attached to the persistent objects. The persistent objects are ordinarily managed by using their names. Although it is necessary to discuss the relationship among naming, scoping, and persistency, we will discuss this problem in another future paper because of the limitation of the space and time. The persistent objects are accessed by referring their names explicitly. If one may consider the environment of Smalltalk-80 as a persistent programming system, the global variables, which include the class definitions, in Smalltalk-80 are persistent objects [Goldberg 83]. O_2 has the named objects for persistency besides the extensions of the persistent types. The policy of the objects based persistency is similar to the ordinary global variables in the programming languages [Lecluse 89]. In the life time of the program, the global variables exist individually. Lisp systems may be considered to have lists based persistency because the global variables usually consist of lists.

We also refer to the cases in that persistency is granted to the several limited types, before proceeding to discuss the requirements for the relation between types and persistency. Since the relation cannot have the pointer as a type for its attribute in the relational database, the schemata are designed to duplicate the values in a column over the different relations. Although this limitation in attribute types is essential to the relational model which is based on the calculus of set, it heavy imposes limitations on constructing a flexible abstraction in the course of the schema design. Semantic data models [Hull 87] [Peckham 88] proposed for the database management systems remove this limitation for represents semantic networks among information directly. Persistency should be orthogonal to types and be designed as an option of type in the persistent programming systems.

As shown in the previous two of the three policies, the constructors of persistent objects and types are strongly associated with each other in many systems. In the persistent programming systems, set is a primitive for persistency, however, the constructors are not limited to only set. The constructors may be not only based on unordered sets but also based on ordered set like Lisp systems. There are various sorts of constructors potentially, e.g. sets, ordered sets, lists, arrays, trees, directed acyclic graphs, graphs, etc. These constructors are described by defining the data structure of the elements of the constructions as the types in the programming languages and are coupled with operations for each constructor into abstracted data types. We propose however these constructors should not depend on the types or the class of objects defined at their creation, but depend on the properties of the some collections of the existent objects as discussed in the subsequent discussions.

If the constructors are separated from the types, it is possible to give persistency to the constructions. In this policy, the sorts of constructors may be limited since constructors cannot be newly defined by using types besides the system defined constructors, although the sorts of constructors may be adequate to ordinary applications. If the constructors are not separated from the types, persistency should be given objects as the third policy, and the types should be defined on any collection of persistent objects, because the constructions of information should be variant dynamically corresponding to the view points. Since an information retrieval makes a new construction of information dynamically, it is required that the construction can be registered as a persistent construction. Consequently, the types which include the construction mechanisms should be based on the existent objects like views. The view (i.e. sub-schema) based constructions of the persistent objects were partially realized on types (i.e. conceptual schema) as options in the systems adopting types based persistency (e.g. relational database systems) [Ullman 88]. The view however is a primitive mechanism for construction based on types in order to describe the dynamic flexible abstraction and to enable person-oriented classifications [Danforth 88].

2.2. Type Mechanisms

The type mechanisms of the open-ended persistent programming systems may become complicated, since the mechanisms should support the applications compiled statically, and also support the dynamic view mechanisms as described in the previous subsection. For realizing both of these requirements opposing to each other, we look back the various sorts of type mechanisms that have been proposed in the research fields of types in programming languages in this subsection. We also construct a direction of type mechanism for satisfying these requirements.

Class based type mechanism:

Since classes are usable for defining the scheme of persistent information, many persistent programming systems adopt this mechanism. Here we should refer to the relation between types and classes. If the types of objects is statically defined, the classes are equal to the types. We however regard types as variant properties of objects corresponding to the view point as explained in the previous subsection. The classes are associated with each other by using the relationship of "is-a" (i.e. inheritances). This mechanism is useful to give some width to a defined type. The class mechanisms however prescribe the scheme of each object to be equal to the definition of the class after creation of object. Consequently, the differentiations of some objects in the same class in the course of evolution [Lieberman 86] or the dynamic views of objects cannot be realized by using the class mechanisms. Although the methodologies of evolution based on the class mechanisms are proposed in several persistent programming systems or database systems [Banerjee 87b] [Penny 87], the rules of evolution seem to be complicated and limited, because the evolutions of objects require dynamic reconstruction of classes in the result.

Ideals model for types:

The *ideals model* regards a type as a set of values [Cardelli 85]. The meaning of that a value has a type is interpreted as that the value is a member of an appropriate set. Consequently, an object can belong to multiple types because the sets may overlap. This model explains the characteristics of the polymorphic types naturally, although the model has some high-order operator, called a *type operator*, out of the model. The relations among types can be replaced by the relations of sets in this model. The assertion "T1 is a subtype of T2" is replaced by the mathematical condition "$T2 \supseteq T1$". The orders of types, namely inheritance mechanisms, are thus represented as the orders of sets.

O_2 adopts the ideals model as an explanation of the semantics of subtypes. FUN, a virtual language prepared for discussing polymorphism by Cardelli [Cardelli 85], has some strong operators on types for enabling the static type checking. This model is thus used for enabling types to have some width in static type checking. This model may also be strong enough to represent the dynamic view mechanisms, since the type as a view can be interpreted as a set in general.

The ideals model founds on the basic types such as Integer, Boolean, or Real. The type of each value is described as an expression consisting of the basic types and type constructors such as function spaces ($\rightarrow$), Cartesian products ($\times$), record types (i.e. labeled Cartesian products), and variant types (i.e. labeled disjoint sums) [Cardelli 85]. The type checking of functions, when they are applied on some values received as their real parameters, uses some type inference rules based on the methodologies of structural equivalence. The equivalence is satisfied by the type of a real parameter is included as a subset in the set, i.e. the type defined for the parameter of the function. In the descriptions of the precise polymorphic types of the functions, the following three qualifiers are usable: the universal qualifiers for expressing the parametric types, the existential qualifiers for representing the abstract data types, and the bounded qualifiers for describing the order between types. It is also possible to combine these qualifiers. The types of the real parameters are replaced with the expressions of the basic types and the type constructors in the result, and are compared with the type of functions. However, the types based on view are not always constructed from the basic types. For example, it is possible to construct the view in that the objects which have same values in their attributes are collected. The view thus can form the type which is defined by some predicates partially, and is not always described directly by the construction of basic types. The type checking in that cases cannot be realized by using only the ideals model.

Type mechanism in Poly:

As an interesting type mechanism which is not based on the basic types, we focus on the type mechanism in Poly [Matthews 85] [Matthews 88]. Poly is one of the languages based on the polymorphic types, and is one of the descendants of CLU [Liskov 77]. In the model of the type mechanism, a type is regarded as a set of operations. The values exist as fundamental entities, and the types are some interpretations for collecting them according to the common operations in these values. This model thus can explain all types including the basic types as abstract data types in which a types is defined with the set of the operations. It is possible to describe the views which are defined according to some operational properties, if the type mechanism of a persistent programming system is constructed on the basis of this model. Although Poly does not have the facilities of evolution, it is implemented as a persistent programming system. This type mechanism does not cope with the relationships among types. The type checking mechanism of Poly therefore bases on the methodologies of name equivalence.

Requirements:

For realizing both of the statically checked types and the dynamically created types, it is necessary to combine the set based type mechanisms such as the ideals model with the type mechanisms based on the properties on the existent objects into a unified type mechanism. The types in such a mechanism may be described as intentional collections consisting of predicates for qualification, or extensional collections enumerating the elements of each set directly. Each type is associated with some operations for describing the properties of the constructions of the collections, such as sets, lists, or arrays. If all elements of a collection belong to a type, the collection can be handled as a subtype of the type, and it is regarded to have the same operations of the type. In order to check types consisting of some collection and a set of operations, the methodologies of structural equivalence should be combined with the methodologies of name equivalence as a result. We present a model adopting this approach of type mechanism in section 3.

It is also required to separate the specifications of types from their implementations for describing more flexible abstraction. Particularly, this separation is necessary for the static type checking. In the static type checking, the partial information about the type is used, where the information of the implementations is not included in the partial information. In the open-ended persistent programming systems, the specifications of the existent types are only used for the type checking as discussed in the next subsection.

2.3. Binding Persistent Information to Ephemeral Applications

There are two policies for the configurations of the persistent programming systems. The configurations are defined by the management of a persistent space where all objects have properties of persistency.

Unified persistent space:

All objects have basically properties of persistency in this policy. The deletions of volatile objects are executed implicitly by system. The techniques of garbage collection are required in the management of persistent storage, as same as Smalltalk-80, or Lisp. This policy is strong enough to construct closed persistent programming systems. Smalltalk-80 has been adopted this policy basically in its environment. Since ORION constructs a unified environment, it adopts this policy. O_2, which supports multiple languages, also adopts this policy, nevertheless the programs written in the languages are statically compiled.

Separated persistent space:

The persistent space is separated from the application space where the objects of application programs exist. Consequently, some mapping mechanisms are required for projecting the proxies of persistent objects into the application space. This configuration is also inevitable in order to achieve an open-ended persistent programming system, since the applications which are written in the conventional

programming languages form the separate application spaces besides the persistent space. Because ODE manages persistent objects separately by using clusters, it is considered to adopt this policy. GemStone also adopts this policy in the interfaces with the conventional programming languages such as C, and Smalltalk-80.

There are also two policies for defining the properties of persistency to objects, besides these configuration policy. These policies concern with the compiler based persistent programming systems in which the programs of the systems are statically compiled, and concern with the systems which adopt the policy of separated persistent space.

Static persistent properties:

The property of persistency of an object is statically defined in the compile time in this policy. Types which are persistent should be declared explicitly in O++ and the pointers which are defined as attributes of persistent objects in the type should also be declared explicitly, if the pointers point the volatile objects.

Dynamic persistent properties:

Any object can have the property of persistency dynamically in this policy. A persistent space which adopts this policy is also called a *persistent pool*. In the case that a system adopts the policy of types based persistency, the name of type is not always equal to the name of a group of persistent objects. Consequently, the persistent objects are referred by another name besides the type name. Although the O++ limits the abilities of persistency of types statically, the objects of the types is managed by using clusters in fact. The clusters are dynamically opened at the run-time of programs. Each cluster then is bounded a particular persistent type. O++ also has the facility for grouping several objects of an identical persistent type by the sub-cluster mechanism. In O_2, an object can become persistent by naming in the course of execution of a program individually, while a unnamed object will be deleted implicitly unless the object is referred from a persistent object.

Cardelli classified the binding policy into three models according to the mechanisms of intern and extern: Fetch-Store model, Load-Dump model, and Commit-Rollback model [Cardelli 88]. In the fetch-store model, a persistent storage is used just as a back-up of objects. It is similar to the combination of the policy of separated persistent space with the policy of dynamic persistent properties. The environment of Smalltalk-80 is equal to Load-Dump model, because this model assumes the single user environment. An object is automatically dumped and loaded by demands. The commit-rollback model is used for sharing persistent objects by multiple applications. It is an extension of the load-dump model. It is similar to the policy of unified persistent space. For discussing more flexible persistency, we refer to some binding mechanisms concerned with persistency

Dynamic binding like Dynamic SQL:

In several advanced cases of dynamic SQL [ISO 89] [ISO 90], an application program receives the information about the scheme of persistent tuples fetched by the application. The application program can deal with the fetched tuples dynamically according to the information. The basic types in persistent information are converted into the types in parent languages, to which some SQL commands are embedded, by the libraries of dynamic SQL. In the open-ended persistent programming system based on a dynamic view as a type, such a dynamic binding mechanism will be required for an interface with the conventional programming languages.

Binding by name:

The persistent objects are accessed by referring to their names. The name of type is equal to the name of type of persistent objects in the type based persistent programming system as shown. In the compile based persistent programming system, the binding will be static.

We summarize the requirements for binding mechanisms. The policy of separated persistent space is indispensable in the open-ended persistent programming systems, however within each system, a uni-

fied persistent space should be achieved for realizing the binding mechanism by name. The policy of dynamic persistent properties is more flexible than the static one. The dynamic binding mechanism like dynamic SQL will be useful when persistent information is accessed by the programs written in conventional programming languages.

We refer to a problem which the designer of object-oriented database will mistake in the last of this section. In the several object-oriented database, on account of supporting the interfaces to the conventional programming languages, the activities of computation is principally executed in the side of ephemeral applications written in these languages. This usage means that the class mechanisms in object-orientations are used for mere schemes of semantic data modeling. The purpose of persistent programming resides in the unified persistent space, and not in the facilities for open-ended. In the unified persistent space, the computation of application should proceed.

3. Our model

We proposed the model for enabling flexible abstraction in [Minohara 90]. The foundation of the model succeeds to the model proposed for persistent programming languages [Minohara 89]. Although there is another concept, Meta (i.e. Machine), in this model besides two concepts as described bellow, we do not discuss the concept directly in this paper. We try to represent how our model satisfies the requirements for persistent programming systems in this section.

3.1. Decomposing the Persistent Objects

We decomposed the concept of an object into two concepts: entity and association. The entity of an object represents its existence, and the associations of the object represent its functionalities. Note that we do not decompose an object into two parts merely. We consider the concept of entity and the concept of association as view points of the concept of object. An single object is considered as an entity in the view point of entity, and it is also considered as a group of associations in the other view point. Here we however adopt the following expression for abbreviation.

an object = an entity + associations

In our model, all objects exist in a unified persistent space. In the persistent programming, the entity expresses the existence of object in the space. Since we adopt the policy of unified persistent space, an object will be removed implicitly unless the entity of the object is named. The naming is the primary mechanism for persistency in our model. The unnamed objects are also not removed, when these belong to the persistent object as its attributes. An association can also have the properties of persistent individually, since it is an entity as noted above. An association persistently defined has a name as well as an ordinary entity.

Associations are used for all sorts of relationship among entities. An entity may be represented as a node in a graph, and an association may be expressed as a directed arc of the graph. Our model may seem to be same as the semantic networks or the semantic data models, particularly functional data model. However, an association is defined as an entity with computation mechanisms. In the functional data model, a function representing the relationship among values is used only for navigating information constructed from these values semantically. It is also possible to use each association for navigating information in our model, since it is one of our total aims to realize an environment where all objects including associations can be managed as operands of an operation. However, we can construct not only information but also computation by using associations.

3.2. Associations

Since the functionalities of an object are expressed by its associations, we discuss the facilities of associations principally in this subsection. Associations are used for describing flexible abstraction. An association is considered as a mapping function from source entities to destination entities, where these entities can be also associations. We call the functionality of an association "association morphism"

generally. An association is conceptually constructed from the description of association morphism and the entities of individual associations. The description of association morphism forms "total association", which any entity coming under the source of association is mapped to some destination, since it is regarded as a surjection. A group of individual associations forms "partial association", which some entity that is not defined in the sources of the association group cannot be mapped to any destination.

An object is formed by attaching individual association to the object. The basic sorts of association morphism defined in the system include the following sorts of association morphism:

entity-of	attribute-of	name-of	reference-of	constraint-of	method-of
source-of	destination-of	association-of	meta-of	version-of	type-of

Several sorts of association morphism are used for constructing the association mechanism itself. Conceptually, an association morphism is not shared by the individual associations of the association morphism. It is copied to each individual association. Although ORION adopts similar approach to our approach, the sorts of association morphism are fixed, and it is not possible to create a new association morphism which has another computation mechanism [Woelk 86]. Note that unnamed objects which are associated with a persistent object through individual associations of "attribute-of" association morphism, are not removed. This sort of association is used for constructing composite objects[Kim 87].

In an object of the conventional object model, the methods which have operations are distinguished from the attributes. In this model, it is possible to regard them as a identical sort of properties since they are linked through individual associations, and is also possible to distinguish according to the sorts of association morphism. The reason why we call association "association", is that the destination entity of an individual association can be accessed only by referring to the name of the association in the point of the source of the association. An association can associate an entity with multiple entities. Such an association is called a "set association". The type mechanism is realized by using set associations as shown in the next subsection.

3.3. Towards constructing type

We enumerate the requirements for the type mechanism in an integrated persistent programming system here: the ability of describing flexible abstraction, the dynamic viewing mechanism, the static description of type by using specifications, and the support evolution based on the alteration of each object. We define a simple language based on our model for explain the construction of type. The language is partially defined only for explain, and has not any strong syntax sugar. We then explain our construction of type by using example of the language.

The syntax of this language is partially defined as follows:

<construction> ::= <definition> | <associating> | <evaluation> | <construction> ,<construction>option
<definition> ::= <name> = <associating> | <name> = <evaluation>
<associating> ::= individual <name> <parameters>option <specification>option (<construction>)option
<evaluation> ::= <name>(<construction>)
<specification> ::= [<source>option >>> <destination>option]
<parameters > ::= [<name definition>]
<name definition> ::= '<name> | <name definition> ,<name definition>option

The option means that the symbol can be omitted. In the <specification>, both of the <source> and the <destination> can be omitted. If they are omitted, the set of whole values in the persistent space is assumed. The <associating> is prepared for creating a new individual association which associates the <source> with the <destination>, while the <evaluation> is prepared for applying an association on the operands (i.e. real parameters) of the association. Since the <definition> can be substituted by the following expression, it is a kind of <associating>:

individual name-of[<name> >>> <associating>], or *individual name-of[<name> >>> <evaluation>]*

We try to describe an object by using this simple language. We then describe Tokyo tower, the tower like Eiffel tower for television, because the beautiful illumination of the tower can be seen from our university at night and Eiffel tower is used for example in O_2. We describe the information of Tokyo-tower by following several attributes: the name, the address, and the height. The address is composed from the following two attributes: the city and the section. We express Tokyo-tower as follows:

```
Tokyo-tower = individual entity-of [>>> (name:,address:(city:,section:),height:) ](
        name = individual attribute-of[ Tokyo-tower >>> "Tokyo Tower" ],
            address = individual attribute-of[ Tokyo-tower >>> individual entity-of(
                city = individual attribute-of[ address >>> "Tokyo" ],
                section = individual attribute-of[ address >>> "Minato" ] ) ],
        height = individual attribute-of[ Tokyo-tower >>> meter( 333 ) ] )
```

We express Tokyo-tower as an entity which has three named individual associations in this description. The entity, which is the destination of the association "address", also has two named individual associations. In this description, some unknown notations appear. The <destination> of the individual association "entity-of" is we describe the tuple which means one entity of a Cartesian product by the notation with parentheses and commas. For example, the tuple consisting of two entity is described as "(,)". The entity of Tokyo-tower is constructed the tuple "(,(,),)". It is possible to refer to the names of fields in a tuple by using the notation "<name>:". The tuple is considered as an entity of labeled Cartesian products. These labels are visible to the outside of the entity of Tokyo-tower. Consequently, these attributes can be accessed from the outside by using the following expression, for example:

```
Tokyo-tower.name
```

In our model, thus, each entity can basically be constructed individually by using individual associations. Consequently, any class system is not required. The specification of an entity is described in the <destination> of an individual association "entity-of" as shown in the previous example. Whether an operation is applicable on the entity or not can be judged by referring the <destination>. In the specification in the <destination>, it is possible to make describe the specification with partial information. The full description of the specification is inferred by system, by referring to the entity bound to the attribute.

In order to make Tokyo-tower a computable entity, we add some method, which displays the color boxes, indicating the illumination of Tokyo-tower, on the screen. We describe the addition of the method as follows:

```
add-visible( Tokyo-tower,
    illumination = individual method-of[ Tokyo-tower >>> → ](
        individual association-of[ >>> ](
            displayColorbox( red, height / 2 ),
            displayColorbox( white, height / 2 ) )))
```

The association "add-visible" adds an entity as a visible attribute to another entity. The expression "→" in the destination of the specification of the "method-of" indicates the destination is an association. This method is described as a total association. Both of the source and the destination in the specification of the association are blank, since this method requires no parameter and returns no entity. The specification of Tokyo-tower will be updated corresponding to this alteration. This method can be used by expressing as follows:

Tokyo-tower.illumination

3.4. Constructions of types based on associations

The facilities of the type mechanism is decomposed three principal functionalities: the creation of a new entity which belongs to a type, the collection of existing entities by using some category, and type checking. Each facility is realized in our model. The creation is executed by using templates. The collection is executed by using view. The type checking is executed by referring to the specification. Although we do not describe precisely the facilities of type checking in this paper because of limitation of the space, the specification mechanisms are used by the interface with the conventional language and are also used when applying some operations on an entity. Consequently, the type is expressed as follows in our model:

Type = Template + View + Specification

Creation by using Template:

Template is a total association for creating objects. While in the class mechanisms, the created object is constrained by a class which has been referred at creation because the class is shared by the created objects, a template referred at creation does not constrain the created object. Consequently, a template is considered as mere a script for creating an object. Muse operating system has been adopting the similar approach [Yokote 89]. The following expression is a template for creating the information of a monument such as Tokyo-tower.

```
createMonument =    individual association-of [ (String,String,String,Integer)
                        >>> (name:,address:(city:, section:), height:) ]
        [ 'aName, 'aCity, 'aSection, 'aHeight ] ( ↑(
            name = individual attribute-of[ Tokyo-tower >>> aName ],
            address = individual attribute-of[ Tokyo-tower >>>
                individual entity-of(    city = individual attribute-of[ address >>> aCity ],
                                      section = individual attribute-of[ address >>> aSection ] ) ],
            height = individual attribute-of[ Tokyo-tower >>> meter( 'aHeight) ] ) )
```

The names of parameters for the association are described in the brackets. The created object will be return by the notation of "↑". Note that "meter" is another template for creating objects of meter types. Although this example has no description on constraints of created persistent object, various sorts of constraints can be attached to the created persistent object. Consequently, the class mechanisms such as one of Smalltalk-80 can be achieved in our model. The following expression describe an example of the usage of this template. The created object becomes persistent because it is named.

```
Marine-tower = createMonument( "Marine Tower", "Yokohama", "Motomachi", 101 )
```

Viewing existing objects:

For implementing a view as a type, the following descriptions are required: the description of a set for extraction by qualifications, the description of a construction for forming the type by properties, and the description of attachments to objects in the view if necessary. The qualifications are described in the sources of the specifications of views, and the type information of the views are described in the destinations. The construction of properties of a view is described in the <construction> of the association for the view. It is also possible to describe the attachments in the <construction>. We describe a view which collects several monuments which height is higher than 100 meter.

```
HigherMonument = individual entity-of[
    [ 'x ][ x in {(name:, address:(,), height:)} and x.height > meter( 100 ) } >>> {(name:, height:)}]
    individual association-of[ {(name:, height:)} >>> HigherMonument ] (
        sorted-print = individual association-of .... )
```

The view named "HigherMonument" collects objects which express the information of monuments, and qualifies them by some predicate. A new method named "sorted-print", which prints all objects corresponding to the qualification in the ascendant order is defined for the construction of the view. Although the borrowing of methods is not described in this example, some borrowing mechanisms are available. Note that the specification in the view has the expression of the collection of entities as its destination, as shown in the previous example, while the specification of the definition of an entity has the expression of an entity in its destination.

4. Conclusion

We discuss the requirements for persistent programming systems in the course of surveying various sorts of proposed systems. Types are described by combining the qualification written by intentional collections and extentional collections with the construction written by facilities of operations. The specification can also be inferred when constructing an open-ended persistent programming system. In a unified persistent space, all objects can be accessed by referring to the name, or by navigating. The type mechanism based on view satisfies these requirements. We presented our model for persistent objects, which based on the concept of entity and the concept of association. By combining various sorts of associations, it is possible to construct not only information of persistent objects but also their computation.

The many problems are left in our researches. We should construct the precise type checking mechanisms based on this model. The evolution mechanism based on each persistent object should be formalized precisely, where the relationship between constraint mechanisms and evolution mechanisms is expressed by a unified rules. These will be realized with the techniques of versioning. The scope mechanism is also an important problem for persistent programming systems. We are designing the prototype system as a persistent programming language which is expressive enough to describe the ordinary programming. In the language, other mechanisms such as meta mechanisms are combined with these mechanism. After the implementation of the language, we will review the experiment of a unified persistent programming system from the practical view point. We may describe another version of this model, on which the reviews will be reflected.

Acknowledgements

The primary author of this paper, Tatsuo Minohara, thanks the members of ICOT database programming languages sub-working group, particularly Mr. Katsumi Tanaka. The discussions of object-oriented database systems in section 2 base on the papers which are surveyed in the sub-working group.

References

Agrawal 89 — R. Agrawal and N.H. Gehani, "ODE (Object Database and Environment): The Language and the Data Model," Proc. of ACM SIGMOD, 1989.

Atkinson 87 — M.P. Atkinson and O.P. Buneman, "Types and Persistence in Database Programming Languages," ACM Computing Surveys, Vol. 19(2), 1987.

Atkinson 89 — M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik, "The Object-Oriented Database System Manifesto," Proc. of DOOD, 1989.

Bancilhon 88 — F. Bancilhon, and et al., "The Design and Implementation of O_2, an Object-Oriented Database Language," 2nd International Workshop on Obejct-Oriented Database Systems, (LNCS, Vol. 334), 1988.

Banerjee 87a — J. Banerjee, et. al., "Data Model Issues for Object-Oriented Applications," ACM TOIS, Vol. 5, No.1, 1987.

Banerjee 87b — J. Banerjee, W. Kim, and et al, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," Proc. of ACM SIGMOD, 1987.

Cardelli 85 — L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," ACM Computing Surveys, Vol. 17(4), 1985.

Cardelli 88 — L. Cardelli and D. MacQueen, "Persistentce and Data Abstraction," Data Types and Persistence, Springer-Verlag, 1988.

Danforth 88 — S. Danforth and C. Tomlinson, "Type Theories and Object-Oriented Programming,", ACM Computing Surveys, Vol. 20(1), 1988.

Date 86 — C.J. Date, "An Introduction to Database Systems 4th Edition Vol.1, Vol.2," Addison-Wesley, 1986.

Goldberg 83 — A. Goldberg and D. Robson, "Smalltalk-80: The language and Its Implementation," Addison-Wesley, 1983.

Hull 87 — R. Hull and R. King, "Semantic Database Modeling: Survey, Applications, and Research Issues," ACM Computing Surveys, Vol. 19(3), 1987.

ISO 89 — "Database Language SQL," ISO 9075, 1989.

ISO 90 — "Database Language SQL2," ISO/IEC JTC1/SC21/WG3 DBL-SEL 3b, 1990.

Kim 87 — W. Kim and et al., "Composite Object Support in an Object-Oriented Database Systems," Proc. of ACM OOPSLA, 1987.

Lecluse 89 — C. Lecluse and P. Richard, "The O_2 Database Programming Language," Proc. of VLDB, 1989.

Lieberman 86 — H. Lieberman, "Using Protypical Objects to Implement Shared Behavior in Object Oriented Systems," Proc. of ACM OOPSLA, 1986.

Liskov 77 — B. Liskov, et al., "Abstraction mechanisms in CLU," CACM, Vol. 20(8), 1977.

Maier 86 — D. Maier and et al., "Development of an Object-Oriented DBMS," Proc. of ACM OOPSLA, 1986.

Matthews 85 — D.C.J. Matthews, "Poly Manual," SIGPLAN Notices, Vol. 20, No. 9, 1985.

Matthews 88 — D.C.J. Matthews, "An Overview of the Poly Programming Language," Data Types and Persistence, Springer-Verlag, 1988.

Minohara 89 — T. Minohara and M. Tokoro, "An Object Oriented Database Programming Language Model," Proc. of IPSJ Advanced Database System Symposium, 1989.

Minohara 90 — T. Minohara and M. Tokoro, "MyAO: A Model for Expressing Persistent Objects," WOOC, 1990.

Peckham 88 — J. Peckham and F. Maryanski, "Semantic Data Models," ACM Computing Surveys, Vol. 20(3), 1988.

Penny 87 — D.J. Penny and J. Stein, "Class Modification in the GemStone Object-Oriented DBMS," Proc. ACM OOPSLA, 1987.

Servio 89 — Servio Logic Development Corporation, "Programming in OPAL," 1989.

Ullman 88 — J.D. Ullman, "Principles of Database and Knowledge Base Systems Vol.1," Computer Science Press, 1988.

Woelk 86 — D. Woelk, W. Kim, and W. Luther, "An Object-Oriented Approach to Multimedia Data-bases," Proc. of ACM SIGMOD, 1986.

Yokote 89 — Y. Yokote, F. Teraoka, and M. Tokoro, "A Reflective Architecture for the Object-Oriented Distributed Operating System," Proc. of ECOOP-89, 1989.