SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2002; **32**:1395–1410 (DOI: 10.1002/spe.489)

Managing schema evolution in a container-based persistent system

J. Baltasar García Perez-Schofield[1,*,†], Emilio García Roselló[1], Tim B. Cooper[2] and Manuel Pérez Cota[1]

[1]*Facultad de Informática, Edif. Politécnico, s/n. Campus As Lagoas, Univ. Vigo, 32004 Ourense, Spain*
[2]*Director of Smarts, Pty Ltd, Level 1, George Street, Sydney NSW 2000, Australia*

SUMMARY

Managing schema evolution is a problem every persistent system has to cope with to be useful in practice. Schema evolution consists basically of supporting class modification and dealing with data objects created and stored under the old class definitions. Several proposals have been made to handle this problem in systems that follow a full orthogonally persistent approach, but, until now, there has not been any proposal to support it in container-based persistent systems. In this paper we describe a schema evolution management system designed for Barbados. Barbados is a complete programming environment which is based on an architecture of containers to provide persistent storage. Barbados does not provide full orthogonal persistence, but, as will be described in this paper, its architecture has several other advantages. Among them is the fact that this model is especially suitable for solving the schema evolution problem. Copyright © 2002 John Wiley & Sons, Ltd.

KEY WORDS: persistence; object-oriented operating system; C++; swizzling; schema evolution

INTRODUCTION

A persistent system is a system in which data structures and data itself persist through executions of any process, as shown in [1]. Under an object-oriented paradigm, the entities we need to make persistent are classes and objects, as explained in [2], and even compiled functions in some systems such as Barbados. A class is probably going to be changed many times throughout the software lifecycle, due to software maintenance. In a non-persistent system, the class would be changed and then the programmer would have to manually write procedures (for example, at load time, converting data in the old format to the new one), in order to adapt all existing stored data to the new class structure.

*Correspondence to: J. B. G. Perez-Schofield, Facultad de Informática, Edif. Politécnico, s/n. Campus As Lagoas, Univ. Vigo, 32004 Ourense, Spain.
†E-mail: jbgarcia@uvigo.es

Received 18 January 2002
Revised 6 May 2002
Accepted 16 August 2002
Published online 27 September 2002

In a persistent system programmers do not have to worry about data storage, therefore it is the system itself which has to cope with the problem of objects being out-of-sync with their class definition. To solve this problem, there are three major approaches described in the literature (for example, in [3]): the system can (a) convert stored objects to its new type immediately or in deferred way; (b) support different versions of the software for the different versions of a class; or (c) simply prevent programmers from changing any data type, compelling users to define new types when needed. This last possibility is unacceptable most of the time.

Related to the previous possibilities, we find two main different ways to cope with this problem in persistent systems like an object-oriented database management system (OODBMS), persistent object stores, or persistent programming languages: schema evolution and schema versioning. Schema versioning, which is explained in depth in [4], consists of supporting multiple user-defined version interfaces of an object, allowing other objects to link to a concrete version of a given object. Although versioning can be found to be interesting for some domains (e.g. see those discussed in [5,6]), we are not going to discuss it in this paper.

When schema evolution is applied, in response to a class definition modification, the system has to modify existing class instances to adapt them to this new definition. This process is generally called *conversion*, as defined in [4]. This change can be done in an eager (or immediate) way, or in a lazy (or deferred) way ([3] offers a detailed explanation of eager and lazy transformations in OODBMSs). The first approach consists of changing all instances of the given class as soon as the class has been changed. The second approach means converting objects only when they are going to be used for the first time.

Although schema evolution has been a focus of active research, it continues to be an unsatisfactorily solved problem as neither of these approaches is broadly suitable. The eager approach has the advantage that, once the conversion is finished, it leaves the persistent store in a consistent state. However, this approach normally requires the database to go through an off-line state, possibly for a long time. The lazy approach is clearly a better one in terms of time and availability, and for that reason it has received much attention in the literature; [7,8] are good examples. A disadvantage of the lazy approach is that instances of newer and older versions of a class can coexist together in time, and the system has to check objects every time they are loaded to convert them if necessary.

In this paper we propose a solution to schema evolution for Barbados, a container-based persistent system. Through a technical discussion, we show that this kind of persistent system is particularly well-suited for solving schema evolution thanks specifically to its container-based structure, explained in [9], which, despite being less orthogonal, as studied in [10], allows us to combine advantages of both the eager and lazy approaches.

The remaining sections of this article are structured as follows. Firstly, Barbados and its persistence model are briefly introduced. Then, we present and discuss our design for supporting schema evolution in Barbados. Finally, conclusions are presented.

THE Barbados PERSISTENT SYSTEM

Within the framework of the container-based persistent model, which is presented in [9], a persistent system called Barbados has been developed as a research prototype. Barbados is an object-oriented persistent programming system, initially presented in [10], integrating persistent object storage and a development environment with a C++-based programming language, a compiler and an integrated debugger. Other systems with facilities similar to Barbados are PJama, introduced with its schema evolution capabilities in [11], JSpin [12], or PerDis, introduced generally in [13]. None of the previous systems offer a working integrated environment. When we focus on the architecture of the system, the most similar system is PerDis, although it is a middleware and not a complete environment. JSpin and PJama are Java-based orthogonal persistent programming systems, thus the architecture and the supported language are different.

However, the main difference between Barbados and other persistent systems is the container-based structure of its persistent storage at the conceptual level. In almost all persistent systems there is always some kind of clustering to improve efficiency. If the fine-grained objects which compose data structures were scattered randomly over the secondary storage, then a persistent system would be unusable because each fine-grained object access would require a disk access and will therefore be unfeasibly slow[‡]. Instead, clustering is used to group objects which are likely to be accessed together when saving data on the persistent store, as studied in [14]. The policy of deciding which objects are to be grouped together can be based on class hierarchy, composition relationships between objects or even time of creation. In orthogonal persistent systems clustering is made at the physical level, thus having the advantage of being transparent to the user. However, this policy has performance disadvantages, because transparent, adaptable and dynamic clustering management (a comparative study of various dynamic clustering techniques is found in [15]) has been shown to be hard to implement in an efficient way, as user intentions have to be anticipated.

In Barbados, in turn, the clustering mechanism is based on user-managed containers, so it is not completely transparent, although a metaphor of directories has been built over the containers. A detailed explanation about the clustering mechanism in Barbados can be found in [9,16]. Therefore, programmers do not have a flat storage model, but rather one based on containers of objects. The abstraction offered is very similar to a file system in which a container behaves like a directory; so in the context of Barbados we employ either terms as synonyms. Programmers are required to explicitly perform the operations of creating, opening and deleting containers, except of course when these operations are hidden inside other commands, such as those built-in for directory management (for example, `cd()`, `mkdir()...`). Furthermore, the user decides which container an object should be stored in, mainly by changing the default directory to be the desired one.

The container is also Barbados' unit of transfer between main memory and the persistent store. A container can be viewed as a large grained object (LGO) which is a collection of fine grained objects (FGOs)—a FGO simply means a class instance, i.e. any normal programming-language level object. FGOs are distributed in non-overlapping containers. A special FGO of each container is called the *root*, because all the remaining FGOs are reachable from it. The *root* always consists of an object of the *directory* metaclass, as stated in [9,16]. This metaclass represents essentially an abstraction of a container, that is, a set of named objects. That is the very same as a directory of a file system, which is basically a set of named files, although objects in Barbados are typically much smaller than files. As in file systems, directories in Barbados can be nested: if a directory A contains a named object also of type *directory*, then that object represents a subdirectory of the directory A. This happens, for example, when the user executes the `mkdir()` system function inside any container.

[‡]Although page caching and other techniques could help greatly in this situation.

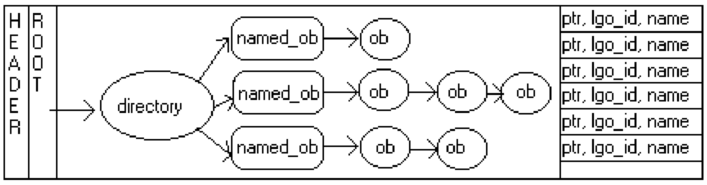

Figure 1. Structure of a container.

The named objects inside a container are the ones which have a public name, which is global and unique when considered as part of the pair *(container_id, object name)*. Objects with a name have an entry in the directory (roughly, each directory is a container): this entry is represented by a *namedobj* object in the container, which points to the true object and holds its name. These named objects are the only ones which can be referenced from another container (through their *namedobj*'s). The rest of the container is not accessible from outside and also, by default, the named objects are only accessible in *read-only* mode when referred from another container. This structure is shown in Figure 1, in which the *named_ob* objects are accessible, while the *ob* ones are not.

One of the purposes of such a structure is to organize the persistent store in some way, preventing the programmer from creating a soup of spaghetti pointers which would create difficulties for an automatic clustering system, as explained in [15]. This structure also prevents data corruption, a problem that is likely to occur in flat persistent stores, as there are no borders among objects or groups of objects, and corruption can therefore be propagated from one group of objects to another, and finally to the whole store. In Barbados, data corruption due to invalid pointers can only happen inside the boundaries of a container, thus limiting the potential damage. Encapsulation of containers is studied in [9,17].

There is another kind of relation among containers which can be found in Barbados: the container-name (CN) Link relation, which happens between two containers. These links can be of three kinds (as shown in [9,16]), although the most interesting case for schema evolution is the *classdef CN Link*, which relates an instance in a container with its *classdef* in another container. This happens when something similar to the following code is executed:

```
cd(/);
mkdir(test);
Barbados> test: container
cd(test);
mkdir(program);
Barbados> program: container
cd(program);
// Container /test/program
class Counter {
       int count;
public:
       int  getCount(void) { return ++count; }
       void reset(void)    { count = 0; }
};
barbados> class Counter {};
// Container /test/data
mkdir(data);
barbados> data : directory

cd(data);
/test/data/Counter c;        // C-N Swizzling relation "data -> program"
        // between the two containers
barbados> c : Counter :
c.reset();
for(int n=0; n<10; ++n)
       cout << c.getCount() << ` `;
barbados>1 2 3 4 5 6 7 8 9 10
```

As a result of all of these instructions, we have two related containers as presented in Figure 2. Technical concepts appearing in this figure, such as '*namedobj*' and so on, are explained in more detail in [10,16]. As can be seen, the code of the application is created in the *program* container (`container_id = 753`), while the data itself (`Counter *c`, i.e. the object of class `counter`) is created in the *data* container (`container_id = 68`). The table of CN swizzling, which is studied in depth in [18], for container *data* (refer to Figure 2) now stores an entry (e.g. {`container id = 753, name = 'Counter', TypeClass`} which is resolved at load time to an ordinary C++ pointer to the class definition object, *classdef*, in another container), which is needed because the object pointed by 'c' has its class definition in container *code* (or 753). The entry in the CN-swizzling table of the container *data* (or 68) also stores an *AbbreviatedClassdef* such as the one below (partially represented in Figure 2).

```
Abbreviatedclassdef: Counter. Referenced in address: 0x00056874
Counter|count|u||
```

An *AbbreviatedClassdef* is just a short description of *classdef*; this way a container referring to a class in another container has a degree of independence, instead of being totally dependent on another container. The influence of abbreviated class definitions in schema evolution is explained in the following sections.

The CN-swizzling mechanism is a powerful and useful one. However, its main purpose is to allow the linking of containers holding data of an application with containers holding the libraries of functions and classes of the application, instead of its use as a general communication mechanism. In the latter case, any container could reference any other in the persistent store, and the whole persistent store would be loaded and unloaded from memory each time Barbados was executed, which is obviously suboptimal.

Also, the introduction of 'containers' might seem contrary to the ultimate goal of persistence, which is to minimize the effort of explicitly moving data between the persistent store and main memory.

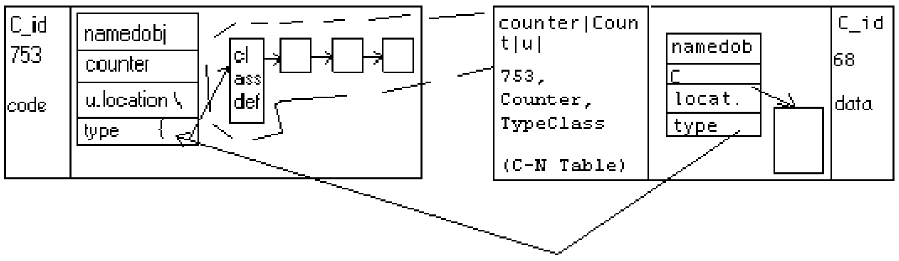

Figure 2. Abbreviated *classdef* relation with foreign class definitions.

We acknowledge that, as a consequence of this design, Barbados does not completely comply to the principles of orthogonal persistence, as defined by Atkinson and Morrison in [2]. However, despite the fact that container-related operations happen at such a *coarse-grained* level, we think it is nevertheless consistent with the goals of persistence, because of the reasons explained in [9,17]. As in other fields (as can be studied in [19]), this lack of orthogonality can be justified on the basis that it achieves other desirable features. That is the case in Barbados, where better efficiency, robustness (which allows us to support a type-unsafe language such as C++, described in [20]) and data protection are the aims of the underlying container-based structure. That is also the motivation for the container-based structure of the Grasshopper persistent operating system (described in [21]). However, Barbados differs from Grasshopper in that Barbados addresses issues of fine-grained object management, whereas Grasshopper leaves that up to application developers, as it is an object-oriented operating system. Something similar happens with PerDis, which is described in [13], in which a library that must be compiled with the persistent project is the interface to the persistent system, giving therefore a much more low-level support than does Barbados.

SCHEMA EVOLUTION IN Barbados

Schema evolution is one of the most important problems still to be solved in the persistence research field. One of the most important references for schema evolution management is O_2, a system described in [22], which has also inspired a quite complete schema evolution mechanism in PJama, presented in [11,23]. One of O_2's strong points is the use of conversion methods, also present in PJama and in Barbados, in the form of conversion functions, that give complete freedom to the user in order to do conversions of all kinds between two versions of a given class. Other systems do not seem to have achieved the same degree of completeness as has O_2. For example, Orion, presented in [24], does not support conversion functions, while JSpin still does not have any implementation of schema evolution (although the JSpin's authors have presented a theoretical study, presented in [25]). Many other researchers have studied the effects of schema evolution in their systems, such as for example Napier, as studied in [26], or, more recently, Oberon-D, commented on in detail in [27].

In this section we describe the schema evolution mechanism that has been added to Barbados. The design of schema evolution support in Barbados has been aimed at providing a ‘semi-automatic’ mechanism, i.e. as automatic as appropriate. We do not provide sophisticated schema-evolution capabilities, because it is likely that no schema-evolution mechanism will be completely satisfactory or include many of the transformations that will be required, as discussed in [3,27]. Another aim was to maximize availability of the system. Therefore, our design premises were clearly biased towards a mechanism which would not require bringing the system off-line, as shown in [7,14]. Finally, we wanted to take advantage of the container-based structure of Barbados compared to flat storages of pure-orthogonal persistent systems.

When considering schema evolution in Barbados we have to take into account the fact that Barbados allows the situation of a container A containing object instances of class X, such that X’s complete class definition can be stored either in the same container A or in a different container B. Thus, we have to consider evolving instances when the container uses a class defined in another container, and a class that changes inside the same container as the data, due to the division of the persistent store presented in [10] and extended in [9].

Detecting schema evolution among containers

As we discussed previously, the situation of classes and objects being in different containers is going to be a very common situation. In fact, this is the most interesting possibility, and it is the basis of the mixed (eager/lazy) approach we follow, so it is explained in the following sections. It is clear that to solve this problem a pure eager approach is not appropriate in this case, since it would require bringing the entire system off-line during the process. Thus, our choice must be some kind of lazy approach.

In this respect, it must be pointed out that a requirement of schema evolution is that the system always knows the type of any piece of data we have. That is not a problem when using an eager approach, as class modification involves immediate adaptation of all objects of this class to the new definition. Therefore, we would only need to store the last definition of each class. However, in a lazy approach, as was intended in Barbados, we would need to keep all possible versions of a class that might have instances in the persistent storage. The concept of a container naturally suggests the following solution to this problem: keep, in a container, a copy of every class of which instances are present in this container. That is the solution adopted in Barbados. However, the copy of a class stored in a container is not a copy of the complete class definition, instead it is what we could call a ‘shallow-copy’; that is, the names, types and offsets of each data member of the class. These pieces of information are called *AbbreviatedClassdef’s* in Barbados, which are presented in [16], and are stored in the CN table, as previously explained. This replication is automatically and transparently managed by the system. Programmers are not aware of this, they only know that a given container has a certain degree of independence despite holding instances of ‘foreign’ classes.

Knowing the type of any piece of data is only the first part of the problem, since at some point we need to determine if this type is out-of-date and then take appropriate action. As we have adopted a lazy approach, this checking will only be done when needed. The commonly adopted solution in orthogonal persistent systems is to perform this check when an object is loaded, as their flat-structure involves a per-object management basis when considering schema evolution.

However, as Barbados follows a container-based architecture, there is a more attractive option: checking all objects of a container at the point when the container is opened. When a container is

Table I. Available possibilities for the *SEvolution* parameter in `OpenContainer()`.

Value	Meaning
SEConvert	Converts all objects in the new container to comply with the new class definition. Quiet mode
SESplit	Creates the old class in the new container, separating these two containers, erasing the CN relation
SEFail	Loading simply fails. The return value of error is E_SCHEVOL
SEAsk	The user is asked about the way to take (the answer is again one of these possibilities, excepting ask, of course)

loaded, all the related containers (i.e. those holding FGOs needed by the first one) are loaded as well, but in read-only mode by default. This includes the containers which store the complete class definitions of the foreign classes. Then, the *AbbreviatedClassdef*'s in each *classdef CN Link* are compared with the corresponding complete class definition. If they are different, the evolution process has to be automatically triggered by the system.

A certain degree of control of the evolution process is available through an argument to the `OpenContainer()` API function. It actually receives three arguments: the *container_id* of the container to be loaded; the mode (*readonly*, *readwrite*) in which it is going to be opened; and a third argument, the one related to schema evolution. The possible values for this argument can be found in Table I.

`OpenContainer()` will be called either interactively or programmatically (for example, through a function which opens a container). The argument used for interactive operations should be *SEAsk*; so, for example, the `cd()` command makes the call to `OpenContainer()` with *SEAsk*, in order to make Barbados ask the user what action to take in the case an out-of-sync relation is found. However, *SEConvert* applies a default conversion without asking the user any questions, provided any class evolution is needed. In the latter case, a conversion function is used. This conversion function can be a default one, generated by the system, or an available, compatible one—if any transformation has been done previously over that class. This is useful in the case when containers are being accessed programmatically instead of interactively.

SESplit means that, if evolution is needed, then the CN link between the two containers is deleted. The CN link exists, as has been seen, as an entry in the CN-swizzling table of a container holding objects, pointing to a container holding their class definition (as shown in Figure 2). If this option is selected, then the system creates a copy of the 'foreign' class, in the same container the objects are in (the *data* container or 68, in the figure). This way, the container holding the objects no longer depends on the container with the class definition (the *code* container or 753). In order to know how to create the class needed, the information stored in the CN-swizzling table entry for that class is used, which holds the *AbbreviatedClassdef* of the class, before evolution.

SEFail means that Barbados will abort any opening if evolution is needed, and an appropriate error will be returned. If the affected class has not been modified but deleted, then the only possible actions would be to '*fail*' or to '*split*'.

In Barbados we have decided to use conversion functions, strongly inspired by the conversion methods which can be found in PJama, which are explained in [11,23], O_2, commented on in [22], and other OODBMSs, compiled in [28]. Conversion functions are explained in detail below; they are simply an opportunity for the programmer to customize the conversion of the affected instances. Here is an example session with Barbados:

```
cd(/);
mkdir(carshop);
 Barbados> carshop : directory;
cd(Carshop);
mkdir(data);
 Barbados> data : directory;
mkdir(bin);
 Barbados> bin : directory;
cd(bin);
#define MAX 250
class car {
private:
              int year;
              bool emission_filter;
              char plate[MAX];
public:
              int price;
              car(char * plate, int age, bool emission_filter = true);
              bool hasEmissionFilter(void) const { return emission_filter; }
              int getAge(void) { return year; }
              char * getPlate(void) { return plate; }
              void Print(ostream &o);
};
 Barbados>class car {}
car::car(char * plate, int age, bool emission_filter = true)
{
              strcpy(this->plate, plate);
              year = age;
              this->emission_filter = emission_filter;
}
 Barbados> car::car: function (ptr to char, int) returning void;
void car::Print(ostream &o) {
              o << plate << '", years: "<< getAge()
                << ", price: " << price
                << ((emission_filter)? "yes":"no") << endl;
}
Barbados> car::Print: function (reference to ostream) returning void;
cd(../data);
/Common/structures/listOfPointers lcars;
Barbados> lcars: list;
```

In this example, there is a *classdef CN Link* between container `/carshop/data` and `/carshop/bin` that is due to the fact that a number of instances of class *car*, which is defined in the *bin* container, are placed in the *data* container. The instances of class *car* are going to be created in this container, pointed by *lcars*, for the whole *carshop* application (although here only a limited view is shown).

The mechanism of schema evolution is going to be explained using this tiny example about a second-hand car shop. Suppose a new law is passed declaring that all cars more than seven years old without internal emission filters will be prohibited in three years. The owner of the car shop decides he is going to get rid of all cars without emission filters, so the `emission_filter` field is no longer required. The way to get rid of the cars is to assign them a very high discount (all cars without emission filters must have their price divided by two, and cars more than seven years old and without emission filter must have their price divided by three). In order to implement these discounts, the *carshop* system will now have two 'price' fields. The `actualprice`, which is private and depends on the value of the car, and the `discountprice`, which is revised year by year and is subject to revision due to many offers. Finally, both prices must be of the *float* type instead of *int* and, as explained, the `emission_filter` field must be removed.

The programmer enters in the `/carshop/bin` and recompiles the cars class as follows:

```
cd(/carshop/bin);
edit(car);
class car {
private:
       float actualprice;
       int year;
       char plate[MAX];
public:
       float discountprice;
       car(char * plate, int age);
       int getAge(void)             { return year; }
       char *getPlate(void)         { return plate; }
       float getActualPrice(void)   { return price; }
       void Print(ostream &o);
};
Barbados>class car {}
car::car(char * plate, int age, float price)
{
       strcpy(this->plate, plate);
       year = age;
       actualprice = price;
}
Barbados> car::car: function (ptr to char, int) returning void;
void car::Print(ostream &o) {
       o << getPlate() << ", years: "<< getAge()
         << ", price: " << (getActualPrice() -- discountprice) << endl;
}
Barbados> car::Print: function (reference to ostream) returning void;
```

While recompiling the class, Barbados finds out that another old class exists, so it fires the schema evolution mechanism. The old *classdef* is preserved, and the container is run over by the system looking for instances of class *car*. As no instances are found in this container, no action is taken. The conversion process is explained below in detail.

Then, the user goes to the '/carshop/data' directory, which stores the list of cars.

```
cd(/carshop/data);
Barbados> class counter doesn't match previous definition:
<C>onvert, <S>plit, <F>ail: C
void convertInstance(car$$_old * oldcar, car * newcar)
{
      newcar->year = oldcar->year;
      strcpy(newcar->plate, oldcar->plate);
      discountprice = 0;
      // newcar->actualprice = 0;
      newcar->actualprice = oldcar->price;
      if (!(oldcar->emission_filter))
            if (newcar->year < 1995)
                  newcar->discountprice = (oldcar->price * 2)/ 3;
            else  newcar->discountprice = oldcar->price / 2;
}
Barbados> Conversion finished.
```

A default conversion function is presented to the user because, after loading the container, the system finds out that the CN relation to the *classdef* in the *bin* container does not match the *abbreviated classdef* (remember it had an emission_filter data member). The lines automatically created in this conversion function (in inverse colours, above) map the compatible data members of the old class with the new one. The user is able to modify this conversion function in any possible way (for example, the user comments out the fourth line, because it does not serve his purpose, while some others are added). This conversion function will be stored for future use in the container holding the class definition, the /carshop/bin container. Once the user completes typing in the conversion function, the container is searched for all instances of the affected class ('*car*', in this case), and the evolution process finishes.

This example shows the main advantage of this approach: containers are converted as they are loaded in memory. Although conversion inside each container is of the eager type, we wait until the container is loaded in memory before converting it. That is why this is a hybrid eager/lazy approach.

We admit that the system is off-line for the user for a while, while the container is being converted. However, a container is a very limited unit compared with the whole persistent system, so we think that this period will be very short.

Conversion functions are explained in detail in a later section.

Detecting schema evolution inside the container of the modified class

As previously explained, in Barbados the user is faced with a directory hierarchy; thus, when defining a class, he first decides the directory and therefore the container where this class definition would reside. Barbados permits one to change a class simply by retyping or editing the class and recompiling it. While compiling, the system detects if a class definition already exists with the same name in the same directory. If so, Barbados will fire the schema evolution process as explained in next section.

So, therefore, in contrast to some other systems, for example those described in [23,28], Barbados does not offer a special language in order to specify changes in classes. We think that the addition of a new, auxiliary, language would be more confusing than this simple solution, which appears to be more natural.

The process would be similar to the one in the previous section, but note that we do not have the '*split*' option available here, as everything happens inside the same container. Therefore, the basis of the schema evolution mechanism *in this case* is the eager transformation, in contrast to the semi-lazy approach in the last section in which containers are converted as they are loaded in memory. Although a container is not limited in size, except for the available amount of virtual memory, typically it will be of the order of kilobytes or megabytes. The schema evolution mechanism will only be applied to the containers loaded in memory, which means that the evolution process can be performed without taking the whole system off-line, but, instead, only a limited set of containers for a limited period of time. This was one of our main design goals.

The mechanism developed seems to be quite efficient, and directly springs from the container-based structure of Barbados. In pure orthogonal systems, the flattened structure prevents us from applying eager schema evolution to a bounded portion of the persistent store. Using eager evolution in these systems implies applying it to the whole store, which also implies putting the store off-line probably for a considerable amount of time.

Converting instances inside containers

Once the system has found out that a given container must have all instances of a given class converted, then the process to follow is always the same. A preliminary step consists of upgrading the container temporarily to read/write access if it is in read-only access mode. If that is not possible—for example, another process is using that container—then the schema evolution mechanism fails.

Once the conversion starts, a lock is established over the container. This means that no other process can access it until the conversion process is finished. The process is based on the already presented

Table II. Table of automatic primitive conversions.

	int	float	double	char	char*
int			assign	assign	atoi()
float	assign		assign	assign	atof()
double	assign	assign			atod()
char	cast	cast	cast		
char*	itoa()	fcvt()	fcvt()	assign	strcpy()

conversion functions and an algorithm which finds all affected instances. Note that the instances of subclasses of affected classes must be converted as well, so an important step carried out before beginning this algorithm consists of defining the affected classes, namely the class detected to have changed, all its subclasses and all classes containing objects of the affected classes as members. Then, the system tries to locate the first affected instance. If that fails, then the conversion process finishes. If that instance is found, then the system searches for an appropriate conversion function in the `/.conv_functions` directory of the container which holds the class definition (not the one which holds the instances). The type information stored in the formal arguments of the conversion functions is enough to store all functions related to a given container using overloading and to find a conversion function suitable for a given class. If a conversion function is not available, an appropriate template for a conversion function is created (as was shown in Figure 2) and prompted to the user. This preliminary template is built using the default primitive conversions, which are shown in Table II. After this step, a conversion function is available in order to apply conversion.

Conversion functions use a special signature for formal arguments, similar to how it is done in the evolution mechanism for PJama, which is described in [11,23]. There are two formal arguments: an instance of the old class and an instance of the new class. Note that they do not pertain to the same *classdef*, although they are two versions of the same class. If the class definition is in another container, then a *classdef* (the 'old' one) must be built from the information in the *abbreviated classdef* entries of the CN-swizzling table. This means that a *namedobj* is also created exactly as if the class had been created here by the user. If the class definition is in the same container, then we already have an old *classdef* and a new *classdef*, as the recompilation of a *classdef* does not eliminate the old one. In both cases, the conversion algorithm is in charge of deleting the old *classdef* when it is finished.

In order to syntactically distinguish between the 'old' and the 'new' *classdef*, a special suffix ('`_$$old`', not expected to be used normally) is added to the name stored in the *namedobj* representing the old class (a similar technique is used in PJama, as shown in [23]). If the old class has data members which are pointers to itself, then they must also be modified in order to make them point to the old class (i.e. the one with the suffix '`_$$old`' in its name). In this way, the conversion function can refer naturally to it as '`classname_$$old`', without needing to modify the compiler in any way. Conversion functions are automatically marked as friend functions of the classes they are expected to cope with. This allows them to have access to all members of the affected class, without any restriction. Once the instance has been located, it is simply created in its new size, copied (the common fields)

```
void convertInstance(a_$$old *old, a *new);

class InstanceIterator {
        InstanceIterator(char *classname);
        void *getFirstInstance();
        void *getNextInstance();
}
```

Figure 3. Conversion API functions and classes.

and finally converted (using the conversion function), without the intervention of the constructors and destructors, which could be dangerous, since in fact the object is not being eliminated, it is just being transformed—although the user is free to call `new` and `delete` in the conversion function. The use of constructors and destructors was discarded since, if important resources are freed in the destructor, for example, there may be a loss of critical information. On the other hand, the algorithm would not know what parameters to pass to a constructor of an object which does not provide a default constructor, as explained in [20].

The table of relocations is used in order to keep track of all objects that have been converted and have had to be moved because the place in which they were was not large enough to fit them. In this table, the size of those objects, their old addresses and the new addresses are stored. Once the conversion is finished, the algorithm runs over the container in order to modify all references to these objects in the table of relocations. Obviously, the algorithm must take into account the fact that C++ allows references to the old object to fall between its old address and its size (i.e. any of its public members can also be referenced).

Finally, this algorithm is safe in terms of reliability: the conversion process will be triggered again if the process is interrupted by an error or a system crash, as the CN-swizzling table would not have been modified, and the need for conversion would be detected again. At the end of the process, the saving—an atomic operation—of the container is forced in order to assure that the whole conversion has been applied. This means that the schema evolution process is executed completely or otherwise it does not have any effect at all.

Advanced conversions

For more advanced uses, which are not covered by the automatic algorithm, the `InstanceIterator` class is available that provides users with a safe way to locate all instances in a given container. The user can then apply to them any possible change, or even delete some of them. As can be seen in Figure 3, the `InstanceIterator` class is provided to the user as part of the conversion API. This is similar to the bulk conversion possibility of PJama, described in [11,23], although the mechanism here is not so automatic. The `InstanceIterator` objects provide a safe way to locate all instances of a given class, but users must write their own function using these objects in order to apply any possible conversion they need, perhaps not covered by the automatic evolution process. This covers our objective of providing a minimum basis of automation, still giving the user, in this way, the possibility of control over the conversion process.

CONCLUSIONS

In this article, Barbados, a persistent programming system based on C++, has been briefly presented; it is similar but different to the completely orthogonal ones. Its main characteristic is that it is based on containers, dividing the persistent store into ‘boxes’ of objects, where these ‘boxes’ are visible to the user through the abstraction of folders (directories) of a file system. We have concluded that containers are a useful abstraction which justifies the need to relax the orthogonal rules for persistent systems.

Also, a design of the schema evolution mechanism for Barbados has been presented. Containers are found to be a successful abstraction for implementing in a very natural way a mixed approach between eager and lazy conversion of objects. Objects of modified classes are converted as the containers in which they live are loaded in memory, while the objects in the same container in which the class lives are converted eagerly. This mixed mechanism is convenient and, to the authors knowledge, this kind of hybrid approach has not been used previously.

Future work about schema evolution in Barbados is to design and implement the migration support mechanism (i.e. when objects are ‘moved’ from one class to another).

REFERENCES

1. Atkinson MP, Morrison R. Integrated persistent programming systems. *Proceedings of the 19th International Conference on Systems Sciences*, Hawaii, 1986; 842–854.
2. Atkinson MP, Morrison R. Orthogonally persistent object systems. *VLDB Journal* 1995; **4**(3):319–401.
3. Cattel RGG. *Object Data Management*. Addison-Wesley: Reading, MA, 1994.
4. Jensen CS, Clifford J, Gadia SK, Hayes P, Jajodia S. The consensus glossary of temporal database concepts. *Temporal Databases—Research and Practice* (*Lecture Notes in Computer Science*, vol. 1399), Etzion O, Jajodia S, Sripada S (eds.). Springer, 1998; 367–405.
5. Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 1996.
6. Rogerson D. *Inside COM: Microsoft’s Component Object Model*. Microsoft Press, 1997.
7. Ferrandina F, Meyer T, Zicari R. Correctness of lazy database updates for an object database system. *Proceedings of the 6th International Workshop on Persistent Object Systems*, Tarascon, Provence, France, 5–9 September 1994, Atkinson MP, Maier D, Benzaken V (eds.). Springer and British Computer Society, 1995.
8. Tan L, Katayama T. Meta operations for type management in object-oriented databases—a lazy mechanism for schema evolution. *Proceedings of the First International Conference on Deductive and Object-Oriented Databases, DOOD ’89*, Kyoto, Japan, Kim W, Nicolas J-M, Nishio S (eds.). North-Holland: Amsterdam, 1989; 241–258.
9. García Perez-Schofield B, Cooper T, Roselló E, Pérez Cota M. Extending containers to address the main problems of persistent object-oriented operating systems: Clustering, memory protection and schema evolution. *Proceedings of the Workshop on Object Orientation and Operating Systems, ECOOP Conference*, Budapest, June 2001. SERVITEC: Oviedo, Spain, 2001.
10. Cooper TB. Barbados: An integrated persistent programming environment. *PhD Thesis*, Basser Deparment of Computer Science, Sidney University, 1997.
11. Dmitriev M, Atkinson M. Evolutionary data conversion in the PJama persistent language. *Proceedings of the Object Oriented Databases Workshop, the European Conference on Object Oriented Programming*, Lisbon, Portugal, 1999.
12. Kaplan A, Ridyway JHG, Schmerl BR. Toward polylingual persistence. *Proceedings of the 9th International Workshop on Persistent Object Systems*, Lillehammer, Norway, 6–8 September 2000 (*Lecture Notes in Computer Science*, vol. 2135). Springer, 2000; 202–217.
13. Shapiro M, Ferreira P, Richer N. Experience with the PerDis large-scale data-sharing middleware. *Proceedings of the Workshop on Persistence Object Systems*, Lillehammer, Norway, 6–8 September 2000.
14. Sousa P, Alves Marques J. Object clustering in persistent and distributed systems. *Proceedings of the 6th International Workshop on Persistent Object Systems*, Tarascon, Provence, France, 5–9 September 1994, Atkinson MP, Maier D, Benzaken V (eds.). Springer and British Computer Society, 1995; 402–414.
15. Tsangaris M, Naughton J. On the performance of object clustering techniques. *Proceedings of the ACM Sigmod International Conference of Management of Data*, San Diego, CA, 2–5 June 1992.

16. García Perez-Schofield B, Cooper T, Roselló E, Pérez Cota M. Barbados: Design of the persistence mechanisms. *Technical Report*, Department of Informatics, University of Vigo, 2001.
17. Cooper TB, Wise M. The case for segments. *Proceedings of International Workshop on Object Oriented Operating Systems (IWOOS'95)*, 1995.
18. García Perez-Schofield B, García Roselló E, Cooper T, Cota M. Swizzling en sistemas basados en containers: Container-name swizzling. *Proceedings of the Octavo Congreso Internacional de Ciencias Computacionales (CIIC'01)*, Colima, México, 2001.
19. García-Roselló E, Ayude J, García Perez-Schofield B, Pérez Cota M. Using orthogonal concurrency principles to effectively separate concerns in COOL's design. *Proceedings of the 8º IEEE Congreso Internacional de Investigación en Ciencias Computacionales*, Colima, Mexico, 28–30 November 2001.
20. Ellis MA, Stroustrup B. *C++. Manual de Referencia con Anotaciones*. Addison-Wesley: Reading, MA, 1990; 541.
21. Dearle A, di Bona R, Farrow J, Henskens F, Lindström A, Rosenberg J, Vaughan F. Grasshopper: An orthogonally persistent operating system. *GH Technical Report on Computer Systems 7,3*, University of Sydney, 1993.
22. Ferrandina F, Ferran G. Schema and database evolution in the O_2 object database system. *Proceedings of the 21st Very Large Databases Conference*, Zürich, 1995.
23. Misha D. The first experience of class evolution support in PJama. *Proceedings of the Third International Workshop on Persistence and Java*, Tiburon, San Francisco Bay, 1–3 September 1998.
24. Kim W, Ballou N, Chou H, Garza J, Eowlk D. Features of the ORION object oriented database system. *Object Oriented Databases*, ch. 11, Brown A (ed.). McGraw-Hill, 1991.
25. Ridgway J, Wileden JC. Toward class evolution in persistent Java. *Proceedings of the Third International Workshop on Persistence and Java*, Tiburon, San Francisco Bay, 1–3 September 1998.
26. Kirby G, Morrison R. A persistent view of encapsulation. *Computer Science 98*. Springer, 1998.
27. Knasmüller M. Adding schema evolution to the persistent development environment Oberon-D. *Technical Report #10*, Institute for Computer Science, Johannes Kepler University, 1997.
28. Brown AW. *Object Oriented Databases* (*International Series on Software Engineering*). McGraw-Hill, 1991.