FLOO: A STRONG COUPLING BETWEEN
EIFFEL LANGUAGE AND O2 DBMS

Robert Chignoli, Jacques Farré, Philippe Lahire & Roger Rousseau
rc@unice.fr, jf@unice.fr, pl@unice.fr, rr@unice.fr

Laboratoire I3S – URA CNRS 1376 & Université de Nice-Sophia Antipolis (UNSA)
250 Av. Albert Einstein - Sophia Antipolis - F 06560 VALBONNE

Abstract:
This paper presents a model of persistency for the Eiffel language and its implementation by FLOO system. This model supports in a transparent way a "contagious" and incremental handling of persistency within a transactional framework. After having defined the context and the constraints of the modelization which had been retained, we present main characteristics of the model which is implemented through routines located in class inherited by all classes of an application. Then, persistent-programming style, which is induced by this model, is illustrated with some basic examples.

Implementation of FLOO system, which uses O2 DBMS as persistent object's server is presented briefly. This choice allows to benefit from capabilities provided by O2 such as an incremental loading, an optimized indexing, its transactional framework using a client/server architecture and its interfacing with other object oriented languages such as C++, for application interoperability. But the strong linking between Eiffel and O2, which is mandatory for an elegant integration of our model in the Eiffel world, means to solve non trivial translation problems for building an O2 image starting from an Eiffel class, and need an adaptation of Eiffel run-time. We end this paper with a first assessement of the project and with perspectives according to 'OJMG'93 works.

Key words:
Object oriented DBMS, Eiffel language, O2 system, contagious persistency, incremental loading, selective queries, navigation, transaction, client-server architecture, interoperability, FLOO, “Business Class” ESPRIT II project.

Object Orientation in Databases and Software Engineering
- 62nd Congress of ACFAS -
Montreal, Canada, May 16-17 1994
Published by World Scientific Publ. Co., 1994

* This research has been financed by the European Esprit II project number 5311 “Business Class”.


1. Introduction

The availability of powerful mechanisms for handling persistency is one of the main problems involved by the emergence of the object technology in the industrial area, especially in business computing. In order to favour this evolution, it becomes important to have in or around industrial object languages some services built upon a “client/server” architecture close to the ones provided by DBMS, such as query facilities and secure access to large collections of objects.

Researches have been undertaken in several directions:

- A first type of solution consists in proposing full O-DBMS, providing, in particular, a specific programming language and libraries of basic components to interface classical or object-oriented programming languages (O2, ONTOS, VERSANT).

- A second approach consists in widening the utilization framework of object oriented language providing libraries of classes acting as persistent repository to the programmers (C++).

- A third direction aims at a better use of object concepts (encapsulation, polymorphism) in order to maximize the integration of services handling persistency. Under this scope, any object may change of status (volatile, persistent) during its life cycle without any explicit action of the programmer. For example, the simple fact of attaching a new object to an object attribute which is itself persistent should automatically imply that the application run-time makes it (by contagion) persistent.

within the framework of the European ESPRIT II project “Business Class” whose general objective is to introduce object technology in business computing area, our task is to design a model of persistency for the Eiffel language. The implementation of its prototype led to FLOO system. After that Eiffel environment solution for persistency has been evaluated (it is of the second type), we chose a model of persistency of the third type which satisfy the following constraints:

- to match with a wide range of situations needing the use of persistency, that is to say being general,

- to favour the reusability of Eiffel components that use persistency, by allowing any type to become persistent, and by leaving in the text of classes as few piece of code as possible for handling persistent objects,

- to preserve reliability (in the sense of correctness, robustness and safety), that is to say:
  - to follow the spirit of Eiffel language, especially its type system and its assertion and exception mechanisms,

---

1 The project started in February 1991, with Eiffel 2.3
2 In French FL and Eiffel are homophones, and O2 is an abbreviation of OO, Object Oriented.
– to propagate automatically the persistent status to objects which depends (by transitivity) on a persistent object: contagious persistency,
– to satisfy the usual integrity and consistency constraints,
– to serialize accesses, making incompatible simultaneous accesses, impossible,
– to control user’s access rights when performing certain kinds of operations.

• to allow the production of efficient applications, that is to say:
  – to minimize exchanges between the main and the secondary memory,
  – to speed up the access to most used object’s characteristics,
  – to slow down as less as possible the handling of volatile objects,
  – to avoid side effects during the program’s execution time, coming from the use of hulk additional data structure.

Section 2 presents briefly the main features of the model which is illustrated by some small examples in section 3. section 4 gives a short overview of the main choices and of the problems which had been solved in the implementation of FLOO prototype. Readers will find in 13 a full description of the model and in 10 a first presentation of the main mechanisms implemented in the current release.

2. Model of persistency

Its objective is to handle the two major types of accesses to objects:

• navigational access within a graph of objects; this access corresponds to the use of classical programming languages;

• selective access to a collection of objects; this is useful for the management of large amounts of data, and corresponds to the privileged type in database languages.

2.1. Modelisation principles for persistency

Our approach relies on the three (golden) rules defined by M. Atkinson 7, to which we add three more in order to preserve the host language’s skills (Eiffel in this case) and to promote efficient solutions.

• Rule 1: independance of persistency. On the first hand, the fact that an object is persistent is independent from the programs that handle it; on the other hand, the source code of a program handling persistent objects should be as independent as possible from the object’s status (volatile or persistent).

• Rule 2: Orthogonality of persistent data types. In line with the general principle of the design of programming language on data type completion, any object, whatever its type, should be able to become persistent and to be handled as such without any limitation.
class HERE
feature
  is_persistent : BOOLEAN is... -- Is current object persistent?
  object_id : STRING is... -- Object reference: address or persistent identifier
...
end -- class HERE

Fig. 1. : Class HERE (1) - Informations dealing with persistence status

- **Rule 3: Orthogonality of persistence management.** The way to introduce and to know about the persistent status of objects is orthogonal to the type system, run-time model and language control data structures.

  All objects referenced by persistent objects remain accessible after the process termination, that is to say they also become persistent. Vice versa, an object may be removed only if no other object references it.

- **Rule 4: Access to persistent objects.** Any persistent object should be accessible through both its identifier (navigational access) and its content (selective or associative access).

- **Rule 5: To meet the host language spirit.** Introducing persistency should not have side effects on other aspects of the language.

- **Rule 6: operation subcontracting.** Operations performed on persistent objects should be subcontracted transparently to the object server which handles persistent object management; we call these operations *treatments in persistent memory*. The same operations on volatile objects should be performed under host language's run-time responsibility.

2.2. Concept of persistent object

In Eiffel 2.3, all classes inherit from class ANY which seems to be the natural location of any new common service. This systematic and automatic inheritance is done via class HERE which is devoted to the reception of a specific site’s services or, a specific version of Eiffel language; so that we located in class HERE services provided by FLOO.

**Status of persistency**  In order to satisfy rules 1, 2 et 3 described above, the fact that a routine handles persistent objects should remain fully transparent to the Eiffel program. Yet, in some special cases, it should be useful to get some informations on referenced objects, especially for the definition of some selection criteria. We introduced some additional features in class HERE to handle those services (fig. 1).

**Concept of obsolete object**  The problem dealing with persistent object’s withdrawal is not only an implementation technical problem (logical removal and/or physical, possible restore) but also a methodological problem:

---

Chignoli et al.  FLOO : A strong coupling between Eiffel and O2 DBMS  Page 4
class HERE

...  
is_still_referenced : BOOLEAN is...  -- Is Current still referenced ?  
is_obsolete(obj: ANY): BOOLEAN is...  -- Is obj obsolete ?  
...

invariant
  not is_persistent implies not is_obsolete  
end -- class HERE

Fig. 2. : Class HERE (2) - Handling of object obsolescence

*Should we enforce the removal of an information which is no longer used?* A volatile object which is no longer referenced during program execution may be removed by a *garbage collector*, but as far as persistent entities are concerned, it is necessary to take into account other aspects linked to the fact that objects may be shared between several applications.

Our model allows to make the difference between an object which is no longer referenced and another which is not useful anymore and which is called *obsolete* (fig. 2).

**Error handling**  When an object is accessed, the system must especially be able to react to the following situations:

1) the object is busy (used by another application),

2) the object is obsolete, that means it is marked as *obsolete* by an application having sufficient access rights on object (Cf. 10, 13),

3) the application does not have access rights on object.

In the first case, the run-time will react according to an indicator handled by the application and accessible through the class HERE. If one object is busy, the application may ask for waiting until object is not busy anymore or to give up.

In the second case, the run-time will trigger an exception in order to tell the application that it may not use an obsolete object. It is necessary to make the difference between an obsolete object and one which is not currently used (*garbage collector* remove not referenced objects only if they are obsolete).

In last case, the trigger will also trigger an exception. Access rights are defined according to the *persistent world* attached to the application.

2.3. *Collections of persistent objects*

In Eiffel, *class* and *type* are not exactly the same, mainly because of genericity. Semantics carried by the effective type of a generic parameter may be more important than the generic class one: for instance, type ARRAY [ WINDOW ] deals more with management of windows on screen than with an array. So that it sounds natural to attach a collection to

1 depending on options set in Eiffel run-time.
class COLLECTION | OBJECT → ANY | export ...

-- Implementation of collection in volatile or persistent memory

inherit

LINKED.SET | OBJECT | -- provide standard services

feature

select(criteria: like\(^\text{\dagger}\) a.criteria; args: ARRAY[ANY]): like Current ...

-- return all object i in Current satisfying criteria(i, args)
do_all (action: like an.action; action.args: ARRAY[ANY]) is ...

-- apply action(i, action.args) to every object i of Current
exists (criteria: like a.criteria; args: ARRAY[ANY]): BOOLEAN is ...

-- is criteria(i, args) for at least one object i of Current?
all (criteria: like a.criteria; args: ARRAY[ANY]): BOOLEAN is ...

-- is criteria(i, args) for all object i of Current?

...

end -- class COLLECTION

Fig. 3. : Methods of class COLLECTION

each type rather than to each class.

The generic term that we retain to point out sets of objects of the same type is collection. It provides a unique framework to handle and have access to a set of persistent or volatile objects of a certain type.

According to persistent objects, three concepts are derived from the one of COLLECTION (fig.3). They are:

- **PCOLLECTION**, allowing to point out persistent objects of a certain type,

- **PSELECTION**, modelizing results of a selective query on a pcollection,

- **PERSISTENT.WORLD**, providing an access to all persistent objects.

After a deep study of different approaches allowing the integration of selections in Eiffel, we have chosen to implement criterias as parametric routines passed to features of collections. With this technique, only one class is necessary to write one or several queries, embedded or not (one criteria may contain other object selections); composition of queries remains possible: results of selective queries return occurrences of type collection. Notions of pcollection and pselection are illustrated in section 3.

2.4. The persistent world

The persistent world represents the set of persistent objects, that is to say the persistent extension of class ANY. It is modelized by following class (fig.4):

\(^\text{\dagger}\) In Eiffel, type "like something" is a syntactical mechanism which forces the type of an entity to be the same than another one of the same class, whatever its possible redefinitions are.
class PERSISTENT_WORLD export ...

inherit

PCOLLECTION | ANY

feature

put_by_key (obj: ANY; symbol: STRING) is ...

-- Record obj under symbol in the persistent world;
-- Make obj and its dependents persistent.
-- If symbol is already in use or if not enough privileges, raise an exception.
put (obj: ANY) is ...

-- Record obj in the persistent world, like put_by_key, but without symbol
remove_by_key (symbol: STRING) is ...

-- Dissociate symbol from recorded object if any or else raise an exception.
-- Object attached to this symbol and its dependents are removed
-- if they are not attached to other objects or symbols.
has_key (symbol: STRING): BOOLEAN is ...

-- Is symbol in use?
item_by_key (symbol: STRING): ANY is ...

-- Object recorded under symbol; void value if no such symbol.

....

invariant

is_persistent

end -- class PERSISTENT_WORLD

Fig. 4. Methods of class PERSISTENT_WORLD
class HERE

... pcollection : PCOLLECTION [like Current] is ...
-- All persistent instances of the object type, accessible to the Eiffel process
  type pcollection (s : STRING) : PCOLLECTION [HERE] is ...
-- All persistent instances of the type s, accessible to the Eiffel process
pw : PERSISTENT_WORLD is ... -- Access to bases of persistent objects
session : SESSION_MANAGER is ... -- Services for transactional management
...
end -- class HERE

Fig. 5. Classe HERE (3): Methods for handling persistency

Operators in the persistent world provide facilities for inserting, reading or removing an object, according to either a symbol associated explicitly by the application (that corresponds to roots of persistency) or one persistent object identifier (POI) generated by the persistent object manager; this allows to improve flexibility and programming capabilities. We also provide features to make an object obsolete within this component.

As for pcollections, it is important for the persistent world to be accessible through any class occurrence, in a very flexible way in order to query the persistent world or to make an object persistent. Hence the persistent world is represented in class HERE by a "once function" \( \| \) (fig. 5, p. 8).

Hence, if an object needs to make itself persistent (1) or to make persistent an object referenced by one of its attributes (an_attribute), it is sufficient to write:

(1) pw.put (Current)
(2) pw.put (an_attribute)

The link between an application and the persistent object manager relies mainly on four features inherited by each class via class HERE (fig.5).

The feature pcollection represents the way, for any object, to have access to the collection of persistent objects matching its type. One may notice that in opposition to 4, we decided that a type persistent collection contains not only its proper occurrences but also those of each conforming type.

The attribute type pcollection deals with the access to persistent collections which contain objects of any type, for instance:

an_object.type.pcollection("LINKED_LIST [ ARRAY [ PERSONNE ] ]")

The attribute session allows to handle the dialog with the O2 server and the synchronization between the persistent world and the application. It is important to note that to have access to one object does not mean to load this object in the process’ associated memory, that an object is loaded in the main memory only when it is mandatory (incremental loading of objects), and that in order to not increase the size of objects, all services of class HERRE are not attributes but functions.

\( \| \) These features allow the implementation of class variables like in C++ (static members) or in Smalltalk.
2.5. *Transactional access*

Within most languages, starting a transaction must be an explicit action; the only language we know which handles transactions implicitly is PCLOS. PCLOS accepts explicit triggering and ending of transaction but by default, a transaction is attached to each loading/update of a persistent object.

We have chosen a similar approach but have adapted it to the Eiffel language. Eiffel allows to export or not a routine, so that a transaction may be defined as the execution of an exported routine. But it may happen that an exported routine calls one or several other exported routines, hence we can get embedded transactions which do not have much interest in FLOO context. In fact our definition is somewhat more precise: a transaction corresponds to the execution of an exported routine only if it is called from a non persistent object and if it applies to a persistent one. Following this approach, any program that may want to use facilities provided by transactions for handling object integrity has nothing to do, except maybe to take into account possible exceptions triggered by the object manager (busy or obsolete object, software failure).

Although transaction’s implicit handling fits very well object integrity management, it reaches its limits as soon as application deals with concurrent access. For instance in one *if* schema, it is not possible to ensure that the result of *if* condition has not changed (because of other processes’ interference) when the *then* or *else* sub-expression is executed. In order to handle such situations, the programmer can have access, via the attribute *session*, to specific features allowing to manage explicitly transactions (*start, finish, abort, validate* ...).

3. Programming style in FLOO

This section presents the programming style induced by FLOO. The proposed examples emphasize especially the *transparency* of persistency and the *integration* of query design. Readers will find in a full presentation of the programming style. The set of examples that illustrates the following sub-section relies on the use of objects of class PERSON represented by the set of attributes that one may find usually for such an entity (fig.6).

3.1. *Insertion of objects within a database*

In FLOO, the notion of *persistent world*, that may be accessed through attribute *pw* and is available within any object, corresponds in O2 to a couple *base/scheme* handled by the object server. Although the mapping between an application and one scheme is frozen when the application is compiled, the mapping between this application and one database is settled at execution time thanks to arguments specifying the *name* of database and its *option* (creation or utilization)**

---

**Moreover programmer has access via attribute *pw*, to features allowing to control (access, modify) the object’s database mapped to the persistent world.**
class PERSON export ...
  feature
    name : STRING; firstname : STRING;
    age : INTEGER; sex : CHARACTER;
    town : TOWN; spouse : PERSON;
    children : LINKED_LIST [ PERSON ];
  Create (p.name: STRING; p.fname: STRING; p.age: INTEGER;
      p.sex: CHARACTER; p.town: TOWN) is do ... end;
  Display is do ... end;
  SetSpouse (p: PERSON) is do ... end;
...
end -- feature

Fig. 6. PERSON class

pl.create ("Dupond", "Jacques", 30, 'm', Nice);
  -- p1 is an object corresponding to the man "Jacques Dupond" living in "Nice"
pw.put_by_key (pl,"dupond");
  -- Now, p1 is a persistent object with the key "dupond"

Fig. 7. Creation of an explicit root of persistency

It is necessary to note that parameters given at launch time will be used only if the application wants to connect to the object server. Typically communication becomes effective between application and object server at the first execution of the feature pcollection, or at the first object insertion; it means that an application which handles only volatile objects never connects to object server. Three techniques are available to insert objects in the persistent world.

Explicit insertion with a key This technique (fig.7) allows to map an object (or a graph of objects) with one name, creating by the way an explicit root of persistency. We use feature put.by.key via attribute pw.

The object pointed out by p1 within the application may be accessed from now on by the persistent name dupond and belongs to pcollection of PERSON.

Explicit insertion without a key This technique (fig.8) allows to do the same thing but in an anonymous fashion through feature put. In this case, as soon as the application’s process ends, the object may only be found (searched) by features of pcollection of person (via one query).

Implicit insertion by contagion This last technique (fig.9) relies on the contagious mechanism. Insertion is performed when a volatile object is attached to one which is persistent.

In previous example, children and spouse of Jacques Dupond become persistent only
p2.create ("Durand", "Jean", 40, 'm', Antibes);
   -- p2 is an object corresponding to the man "Jean Durand" living in "Antibes"
pw.put (p2);
   -- Now, p2 is a persistent object, but without any key

Fig. 8. : Explicit creation of an anonymous persistent object

p10.Create ("Dupond", "Jeanne", 35, 'f', Nice);
p1.SetSpouse (p10);
   -- As p10 is attached to the persistent object p1, p10 becomes persistent too

p15.Create ("Dupond", "Julien", 4, 'm', Nice);
p10.AddChild (p15);
   -- As p15 is attached to the persistent object p10, p15 becomes persistent too

Fig. 9. : Implicit creation of persistent objects by contagion

because they are referenced by a persistent object. It is necessary to note that a status
change does not lead to any immediate exchange with O2 server: run-time reminds status
changes and will handle effective storage at the end of current transaction (Cf. section ).
The symbol dupond allows from now on to have access to full Dupond family members
through object p1 which defines a root of persistency.

3.2. Navigationnal access

We call navigational access an access to an object through doted notation. Within this
context transparency best appears, that is to say run-time support loads incrementally and
automatically the objects pointed out by doted notation, according to needs.

The implementation relies on a set of mechanisms added to the run-time support, which
at loading time of an object obj_1 allows to:

- avoid loading objects referenced recursively by attributes of obj_1, mapping them to
  an answering object which ensures that the true object will be found and effectively
  loaded when it becomes necessary.

- keep in the application's process memory, at any time, only one persistent object copy.

The following example illustrates this situation (fig.10). Note that the notation "?="
corresponds in Eiffel to a "reverse assignment attempt". It succeeds only if type of occurence
to be attached, conforms to the static type of target attribute. If it fails a void value is
returned to the run-time.

p := pw.item_by_key ("dupond");
if not p.void then
   io.putint (p.spouse.town.nb.inhabitants);
end

Fig. 10. Transparent navigational access
a routine is
  do
    p.set.age(p.age + 1);        -- implicit transaction
    session.start;              -- beginning of explicit transaction
    if p.salary < 6.000 then    -- ...
      p.set.salary(p.salary * 1.10)  -- ...
    end;                       -- ...
    session.validate;          -- end of explicit transaction
rescue
  session.abort;              -- end of transaction if an error occurs
end -- a routine

Fig. 11. Explicit use of transactional mechanism

In this example we have chosen to get a persistent object which is a root of persistency using feature item_by_key, then we have gone through this object in order to display the number of inhabitants in the town where the spouse of the person described by the current object lives. On the first hand, it is necessary to note that there is nothing to add in Eiffel source code to implement this action (transparency); on the other hand, only objects mandatory for this display will be loaded in the main memory (incrementality).

3.3. Writing of transactions

If the application is performed in a universe where it is the only one to use the object's database, then it does not need to think about starting or ending a transaction; the run-time handles this automatically. One transaction is triggered as soon as a routine is performed on a persistent object by a volatile one, and ends at the end of the routine. In a universe where several applications share persistent objects, each application must supply run-time for triggering, cancelling or validating one transaction (see fig.11); the applications will use respectively the following features: start, abort and validate of class SESSION.MANAGER, available through "once" feature session of class HERE.

When cancelation is ordered by the application (raising of an exception), it is the application's resposnability to abort the transaction (for instance within rescue clause of a routine). The transactional mechanism influences the persistent object's management; in particular the possible synchronization (validation of transaction) of a persistent object with its image in volatile memory is performed at the end of the transaction.

3.4. Queries

In order to illustrate queries of the FLOO system, we take the most interesting one. selection according to criteria. As the Eiffel language does not allow to pass a function as a routine parameter, selection criteria are described by predicates within classes: We locate also in those classes all the relevant elements for defining criteria, so that they act also as the "context of the query" (fig.12).

As is shown in this example select feature needs two arguments. The first one is the
class SOME CONTEXT
   -- a context with local variables and some criteria
   is_named_Dupond (p: PERSON): BOOLEAN is
      -- Is p named Dupond ?
      do Result := p.name.equal("Dupond")
   end;
   Nice: TOWN;
   is_living_in_Nice (p: PERSON): BOOLEAN is
      -- Is p living in Nice ?
      do Result := p.town = Nice
   end;
   Duponts_living_in_Nice (p: PERSON): BOOLEAN is
      -- Is p named Dupond and living in Nice ?
      do Result := p.name.equal("Dupond") and p.town = Nice
   end;
end -- class SOME CONTEXT

... class SOME_APPLICATION
a_routine (p: PERSON) is
   local
      ctx1: SOME_CONTEXT;
      x, y, z, t: PSELECTION;
   do
      ctx1.Create;
      x := p.collection.select("Is_named_Dupond", one_arg(ctx1));
      y := p.collection.select("Is_living_in_Nice", one_arg(ctx1));
      z := p.collection.select("Duponts_living_in_Nice", one_arg(ctx1));
      t := p.collection.select("Is_living_in_Nice", one_arg(ctx1)).
         select("Is_named_Dupond", one_arg(ctx1));
   end -- a_routine

... end -- class SOME_APPLICATION

Fig. 12. : Query definitions
name of the criterion that will be applied on each object of the persistent collection in order to build the result of query. In order to be able to perform the query in persistent memory, the object’s server needs the image of the class describing the query and its context, but also a persistent occurrence of this class in order to be able to find dynamically the criteria to be applied; this information is called the context and is provided by the second argument of feature select.

Some queries need other arguments, especially those which are embedded. As Eiffel 2.3 does not provide (Eiffel 3 does) routines with a variable number of arguments, we use in this implementation of FLOO, routines which build arrays of arguments with one, two or three elements: one_arg, two_arg, three_arg...

On previous example all selections are performed in persistent memory. The three first selections only need one criteria and are much faster than the fourth one which is designed by composition of queries. All other usual types of queries are available in FLOO (cardinal, exists, all, do.all, do_if, count_if...), and rely on same principles of utilization.

FLOO allows to define more complex queries that require one or several embeddings\(^{\text{11}}\). For instance, in order to build the collection that contains all the towns having at least 500,000 inhabitants with at least 200 of them being more than 90 year old, it is mandatory to apply a query (through feature select) to the persistent collection of TOWN, the implementation of this query using itself the feature select on the persistent collection of PERSON.

4. Description of implementation

Current FLOO’s implementation uses O2 object oriented DBMS\(^{\text{12}}\) as object’s server. This choice allows to get benefit from O2 capabilities dealing with incremental loading, optimized indexing, transactional framework, client-server architecture and interfacing with other object languages such as C++, for application interoperability. But the strong linking with O2 that we retained, in order to integrate in an elegant way our model of persistency in the Eiffel environment, means to solve non trivial problems dealing with the translation of Eiffel classes in O2, and also to adapt the Eiffel run-time. This section gives a short overview of the FLOO global architecture and of the techniques that had been used for implementation purpose. Readers will find in\(^{\text{5,11}}\) a more detailed description.

4.1. Production of an application with the floo command

In order that O2 server understands selections performed by applications, it needs to get within its “scheme of classes” the translation of Eiffel classes; moreover, if we want the selection to be performed in the O2 world (“in persistent memory”), for efficiency purpose, it is necessary to translate in the O2 database language, the criteria and the Eiffel routines that may be associated to selections.

The floo pass manager combines passes of the standard command es and specific passes of the developed environment. Compiling implies to perform successively 7 phases which

\(^{\text{11}}\) An embedded query deals with several persistent collections at a time.
allow to:

- produce (or find) O2 images of application classes and O2 routines to link to the application
- produce binary, especially routines to link the application to the O2 universe
- dialog with the object’s server in order to insert produced O2 images.

4.2. The Eiffel → O2 translator

After studying the possible solutions we abandon the propagation of types into the persistent memory when it is necessary at run-time, and we have chosen a static approach where propagation is managed at compiling time. The translator includes two new passes for O2 translation purpose, and ensures that all classes belonging to the Eiffel system have an image in the persistent world. With this solution, classes are fully propagated in the persistent world (attributes, routines). As the O2 system does not provide genericity except for predefined types, FLOO translator needs to perform a generic instanciation of all parametrized types. It integrates also the pieces of code that are necessary to implement transaction management, format conversion between volatile and persistent objects, incremental exchanges between space of memory associated to Eiffel and O2. But some classes from the ISE kernel library which call external routines in C language, must be translated by hand in O2: ANY, STRING, ARRAY...

In order to handle transaction management, the system adds a call to routines of the FLOO run-time at the beginning and the end of each exported routine allowing to start and end a transaction. If an exception is raised and propagated to the root of the application then the Eiffel run-time asks P0M to abort the transaction (abort.transaction);

4.3. Persistent Object Manager (P0M)

Main functionality of P0M is to store the set of informations which are useful in order to make the application working: schemes produced by our translator, handling of collection of persistent occurrences with O2 databases, interfacing between Eiffel and O2 run-time. In order to implement features of class PCOLLECTION, we relay on O2 programming interface (O2-API).

4.4. FLOO run-time tasks

In order to implement persistency within an application in a transparent fashion, we had to specify new semantics of standard Eiffel run-time features, which had been integrated by SOL company which distributes Eiffel 2.3 in Europe.

All functionalities use a table P_TABLE of persistent object descriptors that may be accessed in a very efficient way through two kinds of identifiers, volatile or persistent. This table allows access to useful informations dealing with persistent objects, whatever
navigational path is used (multiple when object is shared): In which memory is the object, 
to which address, is freeing flag set...

Benchmark on this table show that:

- Overcost according to memory space is very important for a small amount of objects, 
  but is getting smaller as the amount of objects is increasing. It becomes reasonable 
  as soon as 1000 objects are handled.

- the number of indirections necessary to reach objects is not significant until 100 000 
  objects, and remains reasonable even for a large amount of objects (less than 10 
  indirections for 500 000 objects),

- time-consumption within exchanges is reduced until 200 000 objects and remains 
  reasonable after.

For mechanisms dealing with explicit change of status, volatile or persistent, we imple-
mented thanks to p.TABLE, specific algorithms \(^9\) for following situations:

- Implicit change of object to persistent status,
- Persistent object loading,
- Implicit access to a persistent object,
- Implicit update of objects in persistent memory.
- Freeing object from volatil memory.

5. Conclusion

First constraint of our project were to propose a model of persistency satisfying both 
programming principles of software engineering\(^{14}\) and golden rules of object oriented DBMS 
\(^4\). On both aspects, our model satisfies the objectives: all types of objects may become 
persistent, even basic types or those built using the genericity. Persistency is contagious and 
the loading of objects in volatil memory is incremental and often it is transparent Reliability 
of applications is handled with care, by preserving the type system of Eiffel language, its 
assertions and the specific constraints of database, especially transactional accesses.

According to computation efficiency, FLOO implementation provides good results on 
large collections of objects. Although people may regret redundancy between Eiffel and 
O2 worlds dealing first with translation of classes in both worlds and second with the 
representation of persistent occurences. Probably it would be more advantageous to use 
directly a system of lower level than O2 in order to act as object's server, for instance the 
Wisconsin Storage System \(^7\) used by O2.

On a more fundamental aspect, one may ask himself what is missing in Eiffel for being 
an ODL (Object Definition Language) or an OQL (Object Query Language) \(^8\) and to O2 
for being an object language for software engineering. It sounds clear to us that there is
no strong impedance mistmach between those worlds, and we start to work on an object model definition compatible with those of both Eiffel language and ODMG'93, integrating also capabilities for meta manipulations and versionning (see also our approach on reactualization in the IREC project) and the manipulations of lambda-expressions as in the Dylan language.

BIBLIOGRAPHY


Chignoli et al. FLOO: A strong coupling between Eiffel and O2 DBMS Page 17