ORBIT

A bridge between the relational and object worlds

C++/POSTGRES release 2.0

I - Introduction

I.1 - What is ORBIT

ORBIT stands for Object-Relational Bridge by Identification of Tuples.
ORBIT allows to build a bridge between an object oriented system and a relational one. More precisely, the former is any C++ like language, and the latter is an RDBMS.
Any C++ programmer who wants to turn tuples into objects (or objects into tuples) can save a lot of precious time with ORBIT.
First, suppose you want to access to a tuple stored in a relational database, in order to view it, print it, use it, change it, or destroy it. With ORBIT, you can automatically link this persistent tuple to a temporary object (a "standard" C++ object). Any processing of the latter will trigger that of the former.
Second, suppose your OO application generated lots of objects. Unless you used a specific class library, these are temporary. If you don't want your objects to disappear as soon as your application terminates, you just have to use ORBIT, that will transparently grant them persistency.

I.2 - What isn't ORBIT

ORBIT is neither a persistent C++ nor an object oriented database management system.
It is only a bridge, and as such, should only be used to connect an OO application to a relational database.

I.3 - Who needs ORBIT

Any engineer, research scientist, student, or teacher working on the inter-operability of the object and relational models should somehow find out what he looks for, or at least parts of it.

I.4 - Last but not least

ORBIT was developped as main project for a pre-doctorate training course by Eric PONCET, at the Université Pierre et Marie Curie, Paris, France.
Please, before using ORBIT, notice that we provide no warranty as to the fitness of the software for any purpose and in any circumstance whatsoever.
Though remarks about the software may be e-mailed to orbit@masi.ibp.fr, we cannot assist any person or organization in fixing any bug, improving, changing, or upgrading the software.

As you guessed, ORBIT is unsupported software, to be used "as is".

II - Installation

II.1 - Hardware & software requirements

In order to run ORBIT on your system, you must have the following:

. UNIX or the like (current ORBIT version was implemented on a Sun workstation and SunOS 4.1)
. A C compiler ("cc")
. lex & yacc
. A C++ compiler ("CC")
. The POSTGRES RDBMS
. 1 Megabyte free for the complete version (source, library, binary, demo)

If you really want to take advantage of all the features of ORBIT, you should also:

. Have a good knowledge of the relational and object models
. Feel comfortable with C++
. Have a basic knowledge of POSTQUEL, the DML and DDL supplied with POSTGRES

II.2 - POSTGRES

If the POSTGRES RDBMS is not installed on your system, you will have to install it first. POSTGRES is a DBMS under development at the University of California, Berkeley. This is unsupported, public domain software available in source code format on the machine postgres.Berkeley.EDU, Internet address 128.32.149.1.
The POSTGRES team can be reached thru e-mail at post_questions@postgres.Berkeley.EDU.

NOTICE: at the time when this document was written, another machine name was made available.
So, if you can't reach the former, try connecting on s2k-ftp.CS.Berkeley.EDU

Once POSTGRES is installed, you will just have to create or choose an existing POSTGRES user, as well as a relational database as repository for the tuples to be inserted by the demonstration procedure (see `createuser` and `createdb` commands in your POSTGRES documentation).
You must also have an entry in group "postgres", with your UNIX user name (file /etc/group).

II.3 - ORBIT

The installation procedure is quite simple; first, choose a directory in which ORBIT is to be setup; then, at the system prompt type:

```
cd your_directory
mkdir orbit
cd orbit
```

Copy the file "orbit.tar.Z" into the current directory.
You are now ready to uncompress and extract the file hierarchy contained in "orbit.tar.Z"; if you have `zcat`, type:

```
zcat orbit.tar.Z | tar xvf -
```

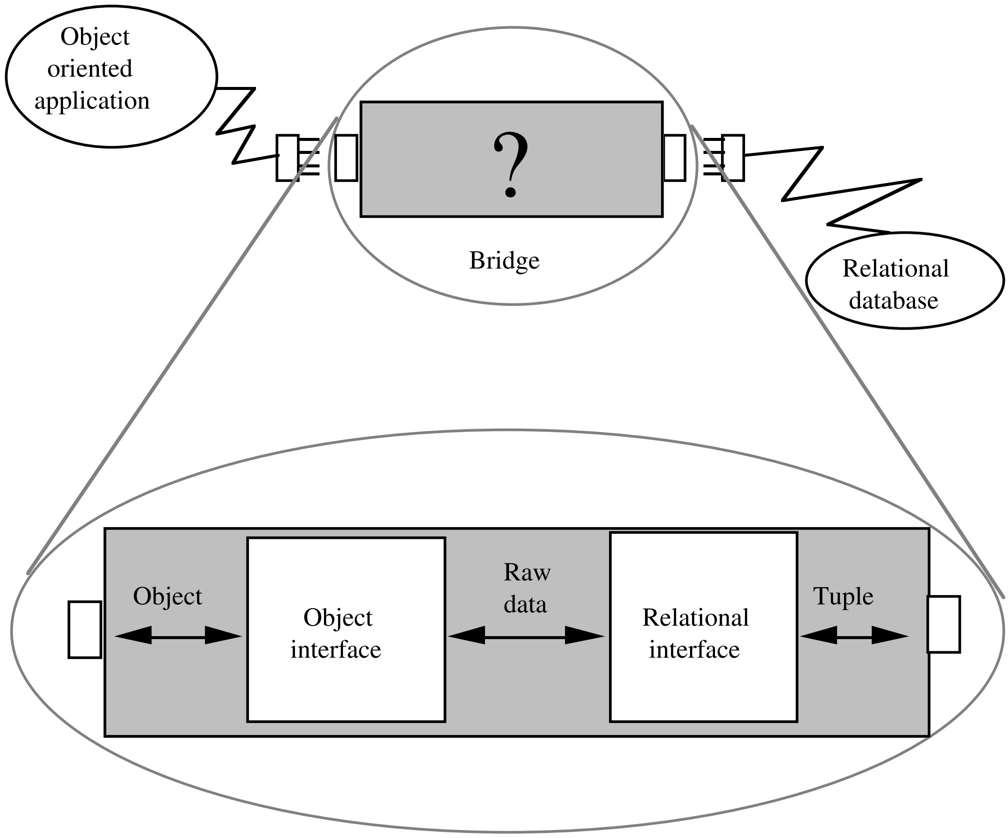
If you don't have `zcat`, you have to uncompress the orbit package, and extract the files it contains:

```
uncompress orbit.tar.Z
tar xvf orbit.tar
```

III - How ORBIT works

III.1 - Basics

If you want to physically interconnect, say, two different networks, you have no other choice but to use a gateway. This kind of device is generally dubbed "blackbox".
You don't know what's inside, but all you expect from is that it works.
ORBIT can also be thought of as a blackbox with two connectors: the one you connect your OO application to, the other you "plug" your relational DB into.



III.2 - Relational interface

This interface is responsible for the translation of data exchanges into relational queries.
For instance, a read access to an object will prepare and trigger a query (SELECT, RETRIEVE, ...), whereas a write access will trigger an INSERT, UPDATE or DELETE.
In order to provide the programmer with optimized access, SELECTs are executed only once for a given object (at the first read access), and INSERTs/UPDATEs/DELETEs are postponed until the very last moment: the end of the relational transaction.

The relational interface consists of a driver, which is structured in 3 modules:

. Transaction management functions, to open, commit, rollback and terminate a transaction
. Data access management, to generate queries on user relational data
. Dictionary access management, to load a relational dictionary into memory

III.3 - Object interface

This part of ORBIT is by far the most tricky.
It allows for simple use and transparency of the relational cookery that is to be done.
To do this, it relies on 2 basic principles: overtyping and code generation.
Overtyping is done by giving basic C++ types a new behaviour, in order to detect changes on any member of an object, and later grant persistency to this change: when the application commits, the new state of the object will be stored into the relational DB. Thus, int, short, long, float, double, char and char* were enriched.
Code generation relies on the use of source code templates that are stored in text files. A specific grammar allows to specify in which places of the source code the relational informations should appear. Thus, templates consists of "normal" C++ source code, with embedded tokens of this specific grammar.

III.4 - Design

The 2-interface design grants a complete independence between the relational and object parts of ORBIT.
The current version of ORBIT can only be used to link C++ to POSTGRES.
But, if you want to link, say, C++ to ORACLE, you'll only have to implement the relational interface specific for ORACLE: no need to review ORBIT's design.
One the other hand, if you'd rather use SMALLTALK with POSTGRES (why not!), you'll just have to rewrite the source code templates that are used for code generation.

III.5 - Implementation

The current version is monolithic.
That is, all the class library that allows for transparency of relational access must be linked with the apllication which uses it.
ORBIT is not structured according to the client/server principles.
All this means that ORBIT currently doesn't care for deadlocks prevention, concurrency and so on.

Clearly, 2 object oriented applications, say *foo* and *bar*, using ORBIT to access to the 2 same RDBs, say *stock* and *account* , could give way to a metabase deadlock.
Just think of the following sequence: foo locks stock and gets it, bar locks account and gets it, foo locks account and must wait, bar locks stock and must also wait. Foo and bar are deadlocked thru their locking of stock and account.

IV - User reference

IV.1 - Building a bridge

In order to build a bridge between an OO application and a relational database, the programmer who uses ORBIT must follow a 2 step procedure.
The first step is the design of the object schema he wishes to define, along with the generation of C++ source code. Once this has been done, the programmer has to use it; that can be done during the second step: modifying and compiling source code, and finally running the bridged relational-OO application.

IV.2 - Setting up a correspondence schema

This step is composed of 4 phases:

. Automatic extraction of flat relational schema
. Enrichment of this flat schema (description of primary & foreign keys)
. Construction of object schema (with a text editor), describing not only the classes but also the correspondence between them and the relations
. Source code generation

IV.3 - Generating, using and compiling source code

Once source code has been generated, it can be modified just like any other source code.
In fact, the generated code contains **structural** (relational classes) and **behavioral** informations (methods for member data manipulation).
This code can be enriched, simplified, and so on.

IV.4 - ORBIT commands

There are 3 commands in ORBIT. Their syntax and function are described below. All of them are **self documented**: just try running from the command shell with no argument.

```
schxtr db_name user_name sch
```

This is the relational schema extractor. If the connection to database `db_name` with `user_name` succeeds, the dictionary is loaded into memory, and formatted into text file `sch`. The text written down this file is based on a specific (but intuitive) grammar.
The programmer then just has to run his favorite text editor to enrich the schema by explicitly describing primary and foreign keys, since this informations aren't stored in the dictionary of most RDBMS.

```
schgen rel_sch obj_sch
```

This command allows the ORBIT user to generate a default object schema. It simply is the raw translation of the enriched relational schema into an identical object schema.
Thus, the programmer only needs to use it if he wants to quickly build a simple bridge.
If he has specific needs, he will have to either define his object schema from scratch (with a text editor) or generate the default object schema and modify it (also with a text editor).
Whatever the solution, the object schema must comply with a specific (but still intuitive) grammar.

`schgen` checks the enriched relational schema `rel_sch` and generates the default schema object `obj_sch`.

`srcgen rel_sch obj_sch`

This is the key command of ORBIT: it brings together and checks the enriched relational and object schema files, and generates the source code, by processing the schemas and template files. Please note that the latter also have a specific grammar, thus allowing for very simple modification of template source code.

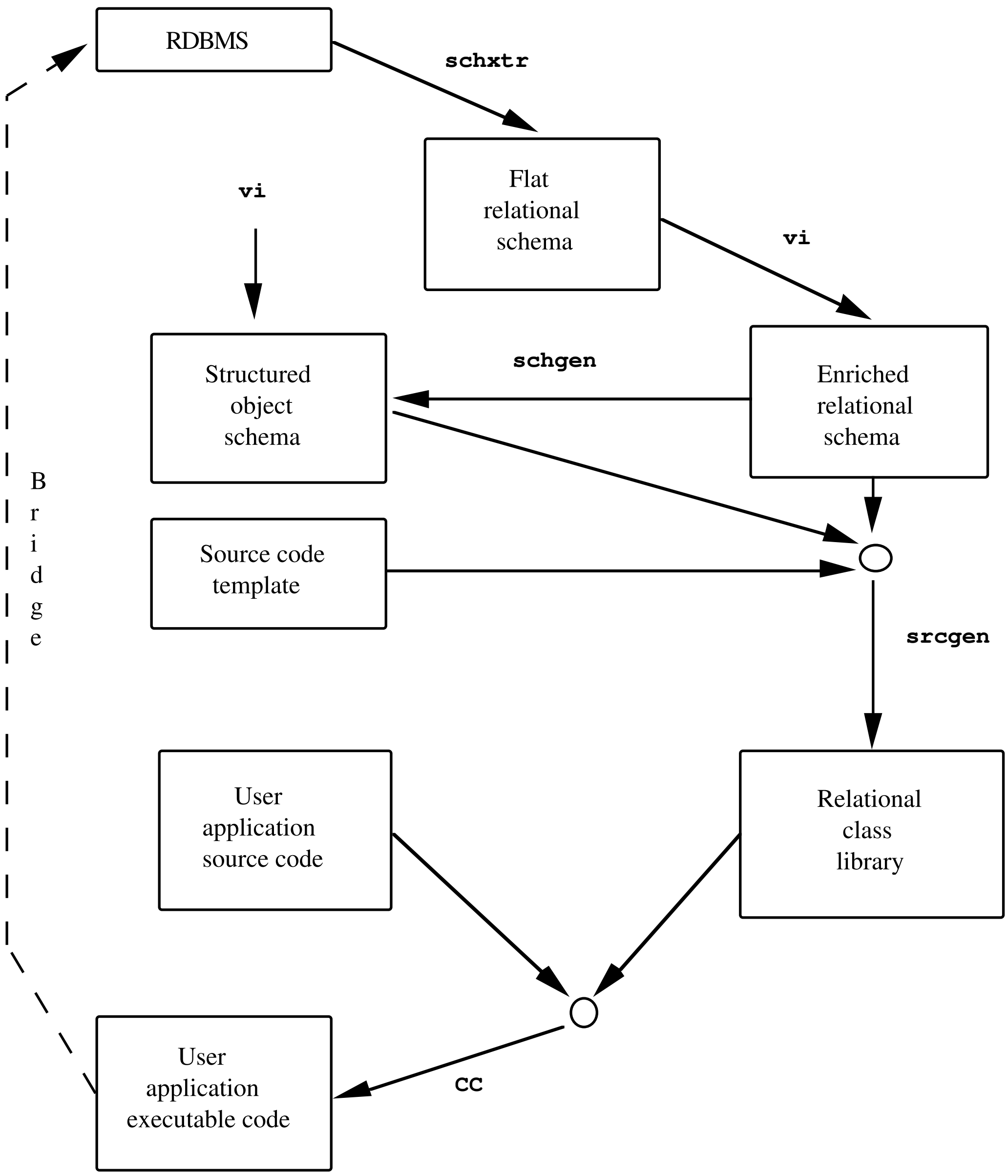
`srcgen` checks the following:

* each relation has got a primary key
* every association of relations by foreign keys is valid (joins)
* every association of classes by members is valid (object networks)
* relation->class and attribute->member correspondences are consistent

In order to make sure that the generated source code is reliable, any error will abort code generation.

`srcgen` brings together enriched relational schema `rel_sch` and object schema `obj_sch`. There are 3 files for each class described in `obj_sch`; their names are made of the classname followed by respectively "`.h`", "`.cc`", "`.mk`".

The following figure is an overview of ORBIT commands, and shows how a relational-object bridge can be built.


RDBMS
schxtr
Flat relational schema
vi
vi
Structured object schema
schgen
Enriched relational schema
Source code template
srcgen
B r i d g e
User application source code
Relational class library
User application executable code
cc

IV.5 - Schema description grammar

Text contained in enriched relational and object schema files must comply with the clauses of the following grammar.

```
schema:
     declarations
     ;

declarations:
     declaration
     | declarations declaration

declaration:
     description ';'
     ;

description:
     /* empty */
     | relation
     | class
     ;

relation:
     Relation name ':'
     Rdbms name ','
     Database name ','
     Owner name
     '{'
       attributes
     '}'
     ;

attributes:
     attribute
     | attributes attribute
     ;

attribute:
     type name attribute_options ';'
     ;

type:
     Char
     | String
     | Short
     | Long
     | Float
     | Double
     ;

attribute_options:
     /* empty */
     | Arrow Key
     | Arrow Foreign name '.' name
     ;
```

```
class:
     Class name ':'
     Relation name
     '{'
       members
     '}'
     ;

members:
     member
     | members member
     ;

member:
     access_option member_type name corresp_options
     member_options ';'
     ;

access_option:
     /* empty */
     | Public
     | Protected
     | Private
     ;

member_type:
     type
     | name '*'
     ;

corresp_options:
     /* empty */
     | '=' name
     ;

member_options:
     /* empty */
     | Arrow Readonly
     ;
```

VI - Example

In order for you to assess the features of ORBIT, we suggest you run an example.
You will be guided to setup a few environment variables, compile ORBIT commands, populate your DB with a few tuples, run the POSTGRES driver demonstration, and a sample OO application.
In case you have a problem compiling programs, read the makefiles; you may have to make a few changes, depending on your C++ compiler ("FLAGS" variable) or on your processor ("OD", "PORTDIR").

If this hasn't yet been done, run the POSTGRES master process:

```
postmaster &
```

Change to your ORBIT directory:

```
cd your_directory/orbit
```

Setup the environment:

```
setenv ORBITHOME your_directory/orbit
setenv ORBITDBNAME your_postgres_db_name
setenv ORBITUSERNAME your_postgres_user_name
```

Create the directory for ORBIT librairies:

```
mkdir lib
```

Compile ORBIT:

```
cd src
make
```

Change to the demonstration directory:

```
cd ../demo
```

Populate your DB with a few tuples (WARNING: the following script destroys a few relations before creating tuples! To avoid losing data, first read thru the populating script before running it !):

```
monitor your_db
\i populate
(lots of messages appear)
\i show
(the tuples created are listed - perhaps not
rectangularly, because of POSTGRES way of
formatting data)
\q
```

Now, you are ready to compile and run the POSTGRES driver demonstration:

```
make pg_demo
pg_demo
```

You can check whether it worked, by displaying the new state of the relations:

```
monitor your_db
\i show
(the tuples are listed)
```

You must now populate your DB again, since the previous demo altered the sample data:

```
\i populate
(lots of messages appear)
\i show
(the tuples created are listed)
\q
```

Compile the sample OO application:

```
make orb_demo
```

`Make` displays the OO schema you will work on, generates the C++ source code of the demo application, and compiles it.
For a better understanding of how the demo and ORBIT work, we suggest you carefully read thru the schema files (".sch" suffix), the demo source code ("orb_demo.cc"), and the generated source code (".h", ".cc", ".mk").

Now, run it!

```
orb_demo
```

At last, check:

```
monitor your_db
\i show
(lots of messages appear)
\q
```