7

Contribution to the Reverse Engineering of OO Applications - Methodology and Case Study

J-L. Hainaut, V. Englebert, J-M. Hick, J. Henrard, D. Roland
Institut d'Informatique, University of Namur
rue Grandgagnage, 21 - B-5000 Namur
jlh@info.fundp.ac.be

Abstract

While database reverse engineering is getting mature, trying to recover the semantics of recent OO applications seems to trigger little interest. The reason is that the problem is underlooked because OO programs *should* be written in a clean and disciplined way, and based on state-of-the-art technologies which allow programmers to write auto-documented code. The paper is an attempt to explain why the reality is far from this naive vision. Mainly through a small C++ case study, it puts forward the main problems that occur when trying to understand actual OO applications. The example is processed through a generic reverse engineering methodology which applies successfully to OO programs, thanks to logical and conceptual OO models that can precisely describe object structures at any level of abstraction. As a synthesis of this case study, the paper discusses the techniques and tool support that are needed to help analysts in reverse engineering the object structures of OO applications.

Keywords

database, data reverse engineering, methodology, object-oriented applications, object-oriented specification, semantics elicitation

Data Mining and Reverse Engineering S. Spaccapietra & F. Maryanski (Eds.)

1. INTRODUCTION

While database reverse engineering is getting mature, as witnessed by the increasing number of conferences on the topic, trying to recover the semantics of recent OO applications seems to trigger little interest. The reason is that the problem is underlooked because OO programs are supposed to be written in a clean and disciplined way, and based on state-of-the-art technologies which allow programmers to write code that is auto-documented, easy to understand and to maintain. It quickly appears that the reality is far from this naive vision, as we will argue in this paper.

So far, the term *OO reverse engineering* has been given two distinct interpretations: namely building an OO description of a standard application and building/recovering an OO description of an OO application. The objective of this paper is to contribute to the solving of the second kind of problems. However, it is worth discussing the goal and problems of both approaches since they share more than we can think at first glance.

1.1 Building an OO description of a non-OO application

According to the first interpretation, a standard (typically 3GL) application is analyzed in order to build an OO description of its data objects and of as many as possible parts of its procedural components. A typical overview of a reverse engineering project following this approach consists in *finding potential object classes and their basic methods.* For example, a COBOL business application based on files CUSTOMER, ITEM and ORDER will be given a description comprising Customer, Items and Order classes, with their associated methods such as RegisterCustomer, DropCustomer, ChangeAddress, SendInvoice, etc. The initial idea is quite simple (Sneed, 1996):

- each record type implements an object class and each record field represents a class attribute;
- the creation, destruction and updating methods (e.g. RegisterCustomer, DropCustomer, ChangeAddress) can be discovered by extracted and reordering the procedural sections that manage the source records;
- the application methods (e.g. SendInvoice) can be extracted by searching the code for the functional modules.

This idea has been supported by much research effort in the last years (Gall, 1995), (Sneed, 1995), (Yeh, 1995), (Newcomb, 1995). Unfortunately, it proved much more difficult to implement than originally expected. Indeed, the process of code analysis must take into account complex patterns such as near-duplication (near-identical code sections duplicated throughout the programs (Baker, 1995)), interleaving (a single code section used by several execution flows (Rugaber, 1995)) and runtime-determined control structures (e.g. dynamically changing the target of a *goto* statement or dynamic SQL). Some authors even propose, in some

situations, to leave the code aside, and to reuse the data only (Sneed, 1996b), among others through wrapping techniques based on the CORBA model. In addition many extracted modules appear to cope with the management of several record types, i.e. with more than one potential object class. This latter problem forces the analyst to make arbitrary choices, to deeply restructure the code, or to resort to some heuristics (Penteado, 1996).

Several code analysis techniques have been proposed to examine the static and dynamic relationships between statements and data structures. Dataflow graphs, dependency graphs and program slicing are among the most popular. In particular, the concept of program slicing, introduced by M. Weiser (Weiser, 1984), seems to be the ultimate solution to locate the potential methods of the record/classes of a program.

1.2 Building an OO description of an OO application

The second interpretation can be read *reverse engineering of OO applications*. Quite naturally, the result will be expressed as OO specifications. The problem is of course different, and fortunately a bit easier, though many standard problems will have to be coped with.

The basic idea is straightforward: the class definitions are parsed and abstracted in a higher-level, generally graphical, OO model*. The schema that is obtained in this way comprises object classes with attributes, inheritance hierarchies and methods. Unfortunately, this schema is far from satisfying. Indeed, most OO applications are written with low-level languages as far as the object semantics is concerned. For instance, Smalltalk, C++ and current OO-DBMS lack many important constructs and integrity constraints that should be necessary to express essential properties of the application domain to be described. Let us mention four of them.

- *Object collections.* Some languages do not propose the concept of set of objects that collects all the (or some) instances of a class. This concept must be simulated by the programmer in ways that are not standardized: *set-of* built-in or home-made constructor, record arrays, files, chains, pointer arrays, bit-maps, etc.
- *Object relationships.* Inter-object associations are not widespread yet, at least in current implementations. They can be implemented in various ways: complex attributes, foreign keys, embedded objects, references (oid) to foreign objects, pointer arrays, chains, etc. Redundant implementations through which bidirectional access is available can be controlled through the *inverse* constraint.

---

* Many tools, such as the integrated development environments, include object browsers that offer this functionality.

- *Identifiers.* An identifier, or key, is an attribute (or set thereof) that designates a property that is unique for each instance of a class. Except in recent proposals (e.g. ODMG (Cattell, 1994)), this uniqueness cannot be declared, but must be procedurally enforced by the updating methods related to the class.
- *Cardinality constraints.* In most OO models, an attribute can be mandatory single-valued (only one value per instance) or multivalued (from 0 to N values*). Defining any more precise constraint is up to the programmer. For instance, asserting that a BOOK has from 1 to 5 authors can be done only by declaring attribute Authors as a *set-of PERSON*, degrading the [1-5] constraint into the more general [0-N]. Here again, the programmer will develop checking code in all the relevant program sections, most generally in the body of the object management methods.

1.3 Motivation

While the relevance of reverse engineering standard applications, typically Cobol/Vsam, Cobol/Codasyl or C/Oracle, can no longer be questioned at the present time, applying this process to *state-of-the-art* programs can seem a bit academic. The short analysis developed above shows that the process can be harder than first estimated. In addition, OO applications can be concerned with the same problems and evolution patterns than typical legacy applications (anyway, there can be C++ and Smalltalk *legacy applications*). We can mention four scenarios in which the reverse engineering of OO applications must be carried out.

- *Redocumentation of OO programs.* There is no objective reason to believe than OO applications have been developed in a more scientific way, with abstract models, CASE tools and complete, consistent and up-to-date documentation, ... than legacy applications. Hence the need to rebuild a correct documentation that will allow the development team to modify, maintain and make the application evolve (Kung, 1993).
- *Translating an application from an OO language to another one* (e.g. converting from to Smalltalk to C++). Since the logical object model of the languages are not identical, recovering an abstract specification of the source application is the most reliable way to build a good quality equivalent target application.
- *Mapping an OO application on a relational database.* This is a very popular reengineering technique to make C++ objects persistent. Since the semantics of C++ and SQL are very different, recovering the conceptual model of the C++ classes is required before generating the semantically equivalent SQL schema.

---

* N standing for ∞.

- *Migrating an OO application to a standard distributed persistant object system such as CORBA or ODMG* (Cattell, 1994). The new standards can express in a declarative way much more semantics than, say, C++ or Smalltalk: relationships, identifiers, inverse, etc. Making these constructs explicit is a mandatory process before building the new schema.

1.4 Objectives and structure of the paper

This paper is a contribution to the process of rebuilding an abstract documentation of the object classes which are declared and manipulated in OO application programs. We base the discussion and our proposals on the DB-MAIN approach and tool that have already proved useful to reverse engineer non-OO data-oriented applications. Section 2 is a short reminding of the components of the DB-MAIN generic reverse engineering methodology. The way object schemas can be represented in an abstract way is shown in Section 3. Section 4 is the development of a case study made up of a short C++ program from which we will try to understand the semantics of object classes. In Section 5, we summarize the main problems that can occur when trying to recover the abstract description of object classes. CASE support is discussed through some representative functions of the prototype DB-MAIN CASE tool (Section 6).

2. A GENERIC DBRE METHODOLOGY

We have proposed a general methodology that can be specialized to the various data models which most legacy systems are based on, such as standard files, or CODASYL, IMS and relational databases. This methodology is fitted to OO applications with few extensions. Since it has been presented in former papers (Hainaut, 1993), (Hainaut, 1993b), (Hainaut, 1994), (Hainaut, 1996b), (Hainaut, 1996c), (Hainaut, 1996d), we will only recall some of its processes and the problems they try to solve, and that will be illustrated in Section 4. Its general architecture is outlined in Figure 1.

The methodology is based on a transformational approach stating that many essential data engineering processes can be modelled as semantics-preserving specification transformations (Section 5.3). Hence the idea that reverse engineering can be (grossly) modeled as the reverse of forward engineering. The model we propose comprises two phases, namely *Data structure extraction*, the reverse of Database Physical design, and *Data structure conceptualization*, the reverse of Database Logical design, that produce the two main products of reverse engineering.

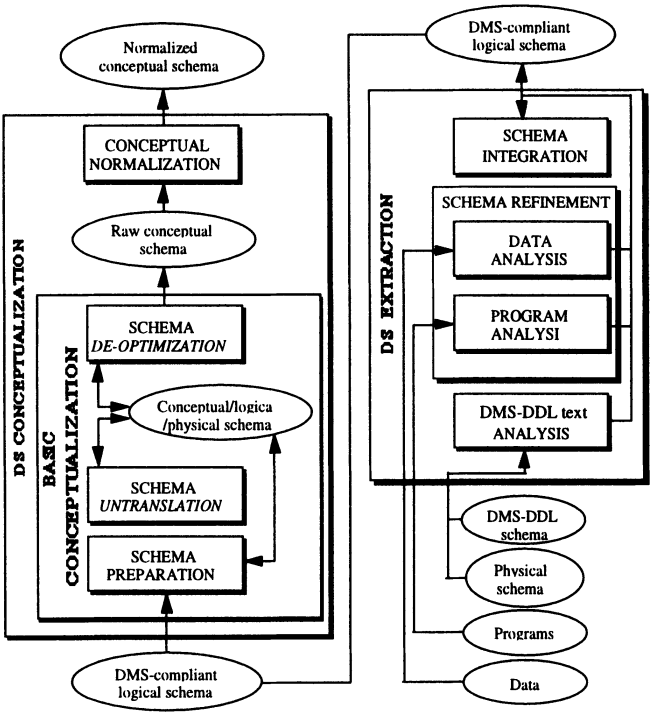

Figure 1 - Main processes of the generic DBRE methodology.

The **Data Structure Extraction Process** consists in recovering the logical schema of the database, i.e. the complete DMS* schema, including all the implicit and explicit structures and constraints. It mainly consists of three distinct sub-processes.

- DMS-DDL text ANALYSIS. A first-cut schema is produced through parsing the DDL texts or through extraction from data dictionaries.
- SCHEMA REFINEMENT. This schema is then refined through specific analysis techniques (Hainaut, 1996c) that search non declarative sources of information for evidences of implicit constructs and constraints (e.g. PROGRAM ANALYSIS and DATA ANALYSIS). This is a complex process that was emphasized rather recently, when analysts realized that many important constructs and constraints are not explicitly translated into DMS schemas, but rather are managed through procedural section, or even are left unmanaged. Hence the many techniques and heuristics proposed in the literature (Andersson, 1994), (Petit, 1994), (Blaha, 1995), (Hainaut, 1996c) to try to recover these implicit constructs. In traditional, non-OO, applications, the analysts will recover structures such as field and record hierarchical structures,

*A Data Management System (DMS) is either a File Management System (FMS) or a Database Management System (DBMS).

identifiers, foreign keys, concatened fields, multivalued fields, cardinalities and functional dependencies.

- SCHEMA INTEGRATION. If several schemas have been recovered, they have to be integrated. The output of this process is, for instance, a complete description of COBOL files and record types, with their fields and record keys (explicit structures), but also with all the foreign keys that have been recovered through program and data analysis (implicit structures).

The **Data Structure Conceptualization Process** addresses the conceptual interpretation of the DMS logical schema. It consists for instance in detecting and transforming, or discarding, non-conceptual structures, redundancies, technical optimization and DMS-dependent constructs. It consists of two sub-processes, namely BASIC CONCEPTUALIZATION and CONCEPTUAL NORMALIZATION.

- BASIC CONCEPTUALIZATION. The main objective is to extract all the relevant semantic concepts underlying the logical schema. Once the schema has been cleaned (PREPARATION) two different problems have to be solved through specific techniques and reasonings: SCHEMA UNTRANSLATION, through which one identifies the trace of DMS translations and one replaces them with their origin conceptual constructs and SCHEMA DE-OPTIMIZATION where one eliminates the optimization structures.
- The CONCEPTUAL NORMALIZATION restructures the basic conceptual schema in order to give it the desired qualities one expects from any final conceptual schema, e.g. expressiveness, simplicity, minimality, readability, genericity, extensibility, compliance with corporate standards (Batini, 1992).

3. REPRESENTATION OF OO CONCEPTS

The methodology produces specifications according to two levels of abstraction, namely logical and conceptual schemas. A *logical schema* expresses the object structures as they are perceived by the user or the programmer of a specific DMS. For instance, we will consider C++, O2 or ODMG logical schemas. A *conceptual schema* is DMS-independent, and expresses the semantics of data structures according to a conceptual model. At this level, we will find OMT, Coad-Yourdon or UML conceptual schemas (Wieringa, 1997).

The DB-MAIN approach is based on a large-spectrum model that encompasses the concepts of most logical and conceptual models used in data engineering. In other words, each practical model can be defined as a specialization of the DB-MAIN generic model (Hainaut, 1989). In the following two sections, we will describe how logical and conceptual OO schemas can be specified in a uniform

way. To simplify the presentation, we will use the graphical notation of the DB-MAIN generic model* to represent schemas at both levels.

3.1 Logical OO schemas

There is a large variety of models at this level, ranging from very basic C++ constructs to recent ODMG and CORBA proposals and OODBMS models. We must represent all of them in as much detail as needed to ultimately recover the semantics of the object classes.

The definition of a class (graphically represented by a rectangle) comprises several parts, namely the class name, the attributes and the methods. A fourth part, dedicated to integrity constraints, will be introduced later on. An attribute is atomic or compound (tuple), single-valued or multivalued (set, bag, list, array).

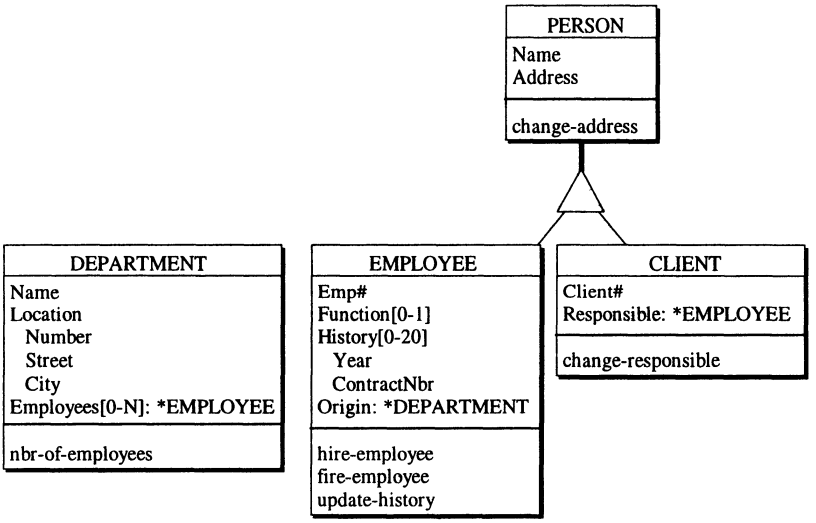

Figure 2 - Attributes and methods of object classes.

A class can be declared a subclass of another one. Each attribute A is given a cardinality [I-J] stating that each parent instance (object or compound attribute value) has from I to J associated values, where J="N" stands for *infinity*. The domain of an atomic attribute is either a basic domain (character, string, integer, real, boolean, BLOB, etc.) or an object class of the schema, in which case it will be called *object-attribute*. For simplicity, the basic domains and the most frequent attribute cardinality (i.e. [1-1]) are not explicitly represented. In addition, only the

* Being intended to express in a uniform way several popular data and information models, these graphical conventions result from trade-offs among the specific graphical representations of each of them. For instance, the representations of an entity type and of a conceptual object class are similar. In the same way, record types and logical object classes are given similar graphical representations.

name of the methods are specified. The schema of Figure 2 includes four object classes. Class **DEPARTMENT** has three attributes, namely *Name* (basic domain, atomic, single-valued, mandatory), *Location* (compound) and *Employees*. The latter is multivalued (set) and its domain is the EMPLOYEE object class, each of its values being an arbitrary large (and possibly empty) set of EMPLOYEE instances. The class has only one method, *nbr-of-employees*. The basic methods such as the constructors, destructors, modifiers and accessors are supposed to be defined, but has not been shown for simplicity. Class **EMPLOYEE** includes two mandatory attributes (*Emp#* and *Origin*), an optional single-valued attribute (*Function*) and a compound, multivalued (list) attribute (*History*) with cardinality [0-20]. *Origin* is a single-valued object-attribute. In addition, this class inherits attributes *Name* and *Address* from class **PERSON**.

These concepts offer a general way to model complex object types built through recursive application of the standard constructors *tuple* and *collection-of* (i.e. set-of, bag-of, list-of or array-of). Though some advanced models comprise the concept of inter-object relationships (e.g. ODMG and CORBA), we will describe it in the conceptual part of the generic model.

Simple object models comprise few (if any) integrity constraints, while more advanced models propose at least identifiers, or keys, and inverse object-attributes.

- *Class identifiers.* A set of attributes form a class identifier (sometimes called *key*) if, at any time, no two objects can share the same values for these attributes. As in relational schemas, a class can have at most one primary identifier (*id*), and any number of secondary identifiers (*id'*). In Figure 3, *Name* is the primary id of class *DEPARTMENT*, while *Employees* is a secondary id. An id is *single-valued* if it comprises single-valued attributes (e.g. *DEPARTMENT.Name*). It is *multivalued* when it is made of one multivalued attribute. In the latter case, no two instances can share each of the attribute values. For example, *DEPARTMENT.Employees* is declared a multivalued id (*id: Employees[*]*), which translates the fact that an employee cannot be employed in more than one department.
- *Inverse object-attributes.* *DEPARTMENT.Employees* and *EMPLOYEE.Origin* are declared *inverse*, indicating that the *Origin* of an employee is the department of which s/he is one of the *Employees*, and vice-versa. Note that the schema indicates that *Employees* is both an identifier (`id'`) and an inverse attribute (`inv`).

We will describe some additional constraints that are commonly found in object schemas, though they sometimes are not explicitly declared.

- *Attribute identifiers.* A compound multivalued attribute A can be given an internal identifier, which is made of a subset I of its components. For each parent instance of A, the values (tuples) of A have distinct values for I. In Figure 3, the *History* tuples of each *EMPLOYEE* instance have distinct *Year* values (*id(History): Year*).

- *Subtype constraints.* The set of subclasses of a given class can be constrained to satisfy set properties: (1) the *disjoint* constraint (symbol D) preventS any two subclasses to share common instances, otherwise, the classes can overlap, (2) the *total* (symbol T) constraint imposes any superclass instance to fall in at least one subclass. Subclasses that are both disjoint and total form a *partition* (symbol P).
- *Other constraints.* Any property that is relevant in other models can be defined on object schemas as well. Such is the case of the *foreign keys*, that can be used to represent many-to-one relationships between two object classes.

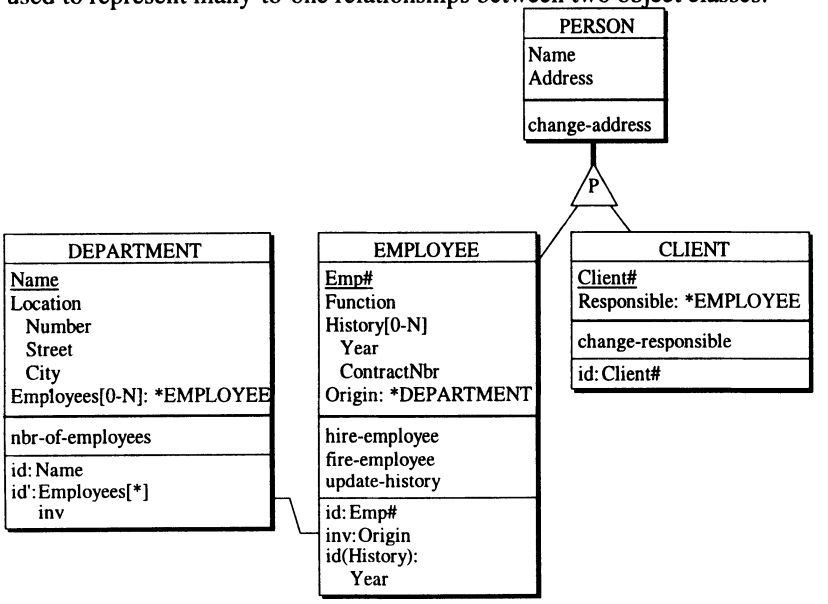

Figure 3 - Integrity constraints: class identifiers, attribute identifiers, inverse object-attributes, subtype constraints.

3.2 Conceptual OO schemas

A conceptual schema is supposed to be DMS-independent, and to offer an abstract view of technical data structures. According to the most popular models (e.g. Coad-Yourdon, Booch, Merise-OO, OMT, Fusion, UML (Wieringa, 1997)), the main aspect of conceptual object models, at least as far as structural aspects are concerned, is the absence of object-attributes and the introduction of relationship constructs (though the latter can be found in some recent logical models). We will discuss how to represent the relationship types. For genericity reasons, we still use the DB-MAIN notation. Though it can appear different from the notation of each OO model, it correctly expresses the main aspects of all of them, and is more than adequate for the purposes of reverse engineering.

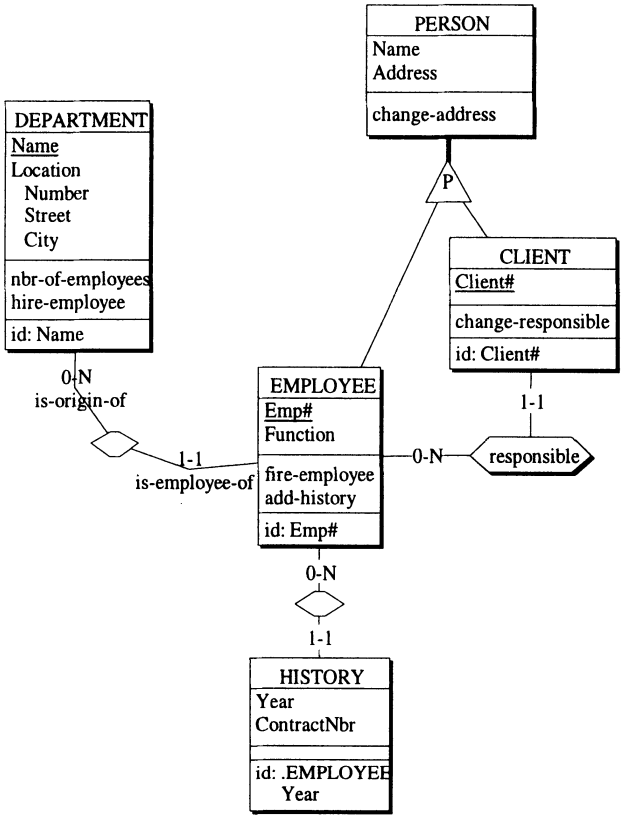

Figure 4 - A conceptual object schema.

A relationship type (rel-type) R has an optional name, comprises at least two roles and can have attributes. A role has an optional name and is taken by one or several object classes. It is submitted to cardinality constraints [I-J] that states in how many (from I to J) relationships any class instance can appear in this role*. In Figure 4, rel-type (*CLIENT, EMPLOYEE*) has name Responsible, while the other two rel-types are unnamed. In the same way, some roles (e.g. *is-origin-of*) are named while others are not. The flexibility of these naming conventions is a necessity for a generic model intended to comply with different operational models.

The identifier of an object class can now comprise attributes, but also roles, as illustrated by object class *History* which is identified by its *Year* value among all the instances associated with one *EMPLOYEE* instance. Such a construct is sometimes called *weak* object/entity type. A N-ary relationship type can have identifiers and attributes too.

---

* Attention: this interpretation, which is compliant with such models as Merise or Batini at al. (Batini,92) is the converse of that of, say, OMT. However, together with the concept of relationship identifier, it encompasses the cardinality concepts of all the other models.

sometimes called *weak* object/entity type. A N-ary relationship type can have identifiers and attributes too.

4. A SHORT CASE STUDY

We will discuss the main concepts developed so far through a small case study based on the C++ program presented in Appendix, the goal of which is the management of information concerning customers, orders and products. This application is incomplete and unrealistic (data are not saved on program closure for instance), but it is sufficient to illustrate both the kinds of problems that actually occur in practice and the reasoning that can solve them. Most of the processes mentioned in Section 2 apply, and will be discussed in some detail.

4.1 The DMS-DDL text ANALYSIS process

The C++ analyzer recognizes the class definitions: class name, superclasses, types, attributes and methods (Figure 5). Pointer to class instances (e.g. `customer *next`) are abstracted as object-attributes (e.g. `next: *customer`). Arrays[I] are expressed as array-multivalued attributes with cardinality [I-I]. All the attributes are supposed to be mandatory. This process is a mere representation translation and uses mere parsing techniques. In particular, it does not rely on any reverse engineering knowledge.

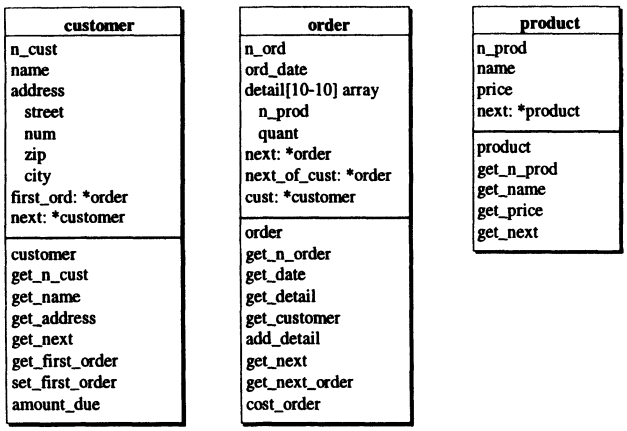

Figure 5 - The first-cut logical schema.

4.2 The SCHEMA REFINEMENT process

In most situations, the resulting schema will be too coarse, mainly due to the lack of expressive power of the language. The refinement phase is intended to recover the implicit constructs and constraints that are buried in the programs. Among the

numerous implicit structures and constraints that can be sought (Hainaut, 1996c), we will concentrate on six of the most important ones. In addition, we will base this process on program analysis, thus ignoring the other sources of information such as the data, the user interface, program execution and the documentation (whatever its state).

One of the frustrating aspects of reverse engineering is that one cannot guarantee 100% recovery of the specifications. As it is true for any knowledge extraction process, where quality of the final product is a (hopefully increasing) function of the quality and completeness of the sources of information, and of the effort we accept to put in the process.

A. Optional attributes

In C++, all the class attributes are considered mandatory. By analyzing the way each attribute is initialized, processed and used, we can discover whether it accepts void, null or default values, which most often stand for *absence of value*. As an example, we will examine the attributes of `customer` through the behaviour of its constructor `customer::customer` [line 108] which is where essential integrity constraints should be monitored.

Observations.

- `n_cust` receives the value of input argument `n_c`,
- `name` receives the value of argument `n`,
- the components of `address` receive the values of those of argument `adr`,
- `next` receives the value of `customers`
- `first_ord` is set to null.

The analysis is not complete since we must understand the history of each of the source variables. Further analysis leads to learning that:

- the constructor is called from one program point only, namely procedure `new_cus` [line 263]; in its body, we observe that all input arguments of the constructor are set to non-empty values yielded by the user, except for `zip`, for which no checking is performed;
- `customers` is a global variable initially set to NULL,

Conclusions. All the attributes of class `customer` are mandatory (cardinality [1-1]) but three of them, namely `address.zip`, `first_ord`, and `next` (cardinality [0-1]). By similarly analyzing the origin of the attributes values in the constructors we can state the cardinality of the other attributes of the schema.

B. Exact cardinality of multivalued attributes

C++ arrays are given a number of cells, but there is no way to declare how many cells can, and must, receive actual values. That is the case of attribute `order.detail[10]`. The extraction process can only abstract such constructs

as an attribute `detail[10-10]array`. Obviously, this cardinality must be refined. Manipulation of the array elements can be found in the procedure `new_ord` that introduces a new order, and in the methods it invokes, namely the `order::order` constructor and method `add_detail`.
Observations.

- in `new_ord`, `detail[*].n_prod` and `detail[*].quant` are set to 0 through calling the constructor,
- then, `add_detail` is invoked as many times as there are non-zero `n_prod` values entered by the user; entering no details is a possible event,
- as expected, `add_detail` does not allow more than 10 product numbers to be specified.

Conclusion. The exact cardinality of `order.detail` is `[0-10]` instead of `[10-10]`.

C. Class identifiers

It is natural that each major object class should be given an explicit identifier, allowing users to designate, e.g., a specific customer or a definite product. Name patterns and domain knowledge are of particular help in this quest, but we will use program pattern analysis, specifically in the constructors, to find possible class identifiers. We will concentrate on customer class.

Observation. The constructor includes an *emergency exit* [line 109] through which no `customer` instance is created. This exit is triggered by a positive answer to the invocation of `find_customer` function. The latter returns the customer `oid` (physical address) of the first customer instance with attribute `n_cust=n_c`, i.e. a customer that has the same `n_cust` value as that one tries to introduce.

Conclusion. `n_cust` is an identifier of object class `customer`.
Through the same analysis, we find the primary identifiers of classes `order` and `product`.

D. Attribute identifiers

For any multivalued compound attribute, the question of whether a uniqueness constraint is enforced must be asked. That is the case for `order.detail`, the management rules of which are concentrated in method `order::add_detail`.

Observation. The while loop terminates when either (1) all the `detail` cells have been examined without success, or (2) the first cell with `n_prod = 0` has been found, or (3) the first cell with `n_prod = n_p` has been found. Then, a new tuple is inserted when an empty cell has been found, i.e. in case 2. In summary, a new tuple is inserted when the array does not include another tuple with the same `n_prod` value.

Conclusion. Attribute `n_prod` is unique in the set of `detail` tuples of any `order` instance, and must be declared an identifier of attribute `detail`.

E. Non-set multivalued attributes

Due to the lack of expressive power of standard programming languages, including C++, small sets of values most often are implemented as arrays, such as `order.detail[10]`. By examining how the elementary values are processed, we can learn whether the position or the ordering of these values are significant.

Observation. The `detail` attribute is managed in function `add_detail`. We already know that it represents a *set* of values, and not a *bag*. Obviously, no meaningful ordering seems to be maintained. In addition, the other program sections use indexing of the array elements only to get them, and there is no apparent meaning associated with this index.

Conclusion. `order.detail` is just a set of tuples where the element position and ordering are immaterial.

F. Foreign keys

Domain knowledge suggests that some links should exist and be maintained between `order` and `product` instances. The examination of the methods `order::add_detail` and `order::cost_order` gives us the key.

Observation. The most obvious observation relates to attribute names and domains: two attributes, namely `order.detail.n_prod` and `product.n_prod`, happen to share their names and domains. In addition, one of them is an identifier. In the method `add_detail`, a `detail` tuple is inserted only when the value of input argument `n_p` identifies a `product` instance. Considering that this method is the only way to add details, we can be sure that there is no `detail` tuples without a matching `product` instance. Another evidence can be found in method `cost_order`. To compute the cost of the order, the body of the method finds the `product` instance referenced by each `detail` tuple. We observe that the `price` of this instance is asked for without worrying about its existence, which seems to be taken for certain. We conclude that each `detail` tuple is guaranteed to have a matching `product` instance, unless the program is wrong.

Conclusion. The component `n_prod` of `order.detail` is a foreign key to object class `product`.

G. Access structures

Unlike higher level data managers, C++ offers no explicit constructs to provide programmers with instance collection and traversal techniques. The programmers

have thus to implement technical structures to maintain instance collections and to access successive instances of a class. Chaining is one of the most popular technique to implement ordered set of instances. A chain comprises an external head pointer that yields the first element of the chain, and next pointers that yield, for each chain element, the next element, if any. Typically, the next value of the last element is null.

Observation. In a C++ class structure, chains generally are implemented through instance pointers that are abstracted as attributes defined as `attribute-name[0-1]: *class-name`. The example schema comprises five such attributes*, whose behaviour has to be examined in detail. Let us consider attribute `next` of class `customer`. As expected, this attribute is processed in the constructor of its class, and it appears that it is used to chain all the customer instances. The global variable `customers` acts as chain head. This hypothesis is confirmed by the analysis of application procedure `list_cus`, which is based on a chain traversal loop. The analysis of the attributes `order.next` and `product.next` leads to similar conclusions.

Through the same approach, the pointers `customer.first_ord` and `order.next_of_cust` also appear as implementing chains whose head is in a `customer` instance, and chaining `order` instances.

The random way instances are inserted in the chains suggests that instance ordering is immaterial, and that they implement unstructured sets only.

Conclusions. There are two kinds of chains, those which merely implement the collection of instances of each class, and those which implement access of a list of `order` instances from each `customer` instance. The first kind of chains can be discarded since the abstract OO model we use includes the notion of instance set of classes. The second kind of chains seems more prone to support semantics, and must be kept. However, for reasons that will soon appear, these chains need to be further processed.

First we replace it with another equivalent construct, namely multivalued object-attribute `orders`. We observe that an `order` instance could not be inserted in more than one such chain, a property that translate into the following constraint: an `order` instance cannot appear in the `orders` set of more than one `customer` instance. In other words, `orders` is a *multivalued secondary identifier* of class `customer`.

When a class A includes an object-attribute with domain B, it is non unfrequent that class B includes an inverse object-attributes with domain A. This could be the case with class customer with attribute orders and class order with attribute customer. The analysis of procedure new_ord shows that the value of attribute cust of the current order instance is the address of the customet instance in the

---

* To be quite precise, we should have proved that each of them is a secondary identifier of its class.

chain of which this order instance is inserted. Consequently, attributes `customer.orders` and `order.cust` must be declared *inverse*.

The final logical OO schema is presented in Figure 6.

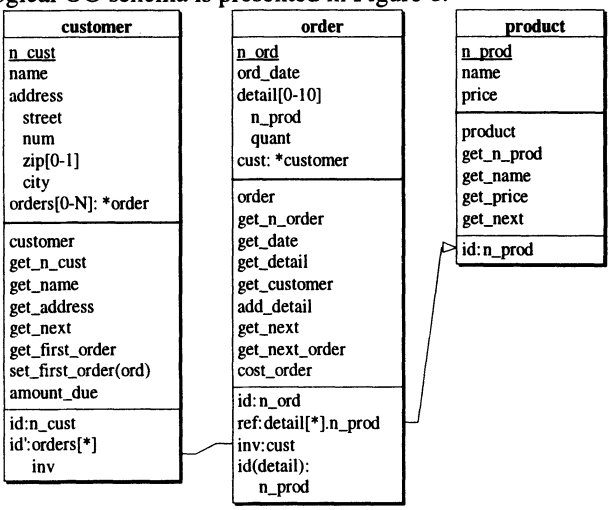

Figure 6 - The refined logical schema. The multivalued foreign key detail[*].n_prod is symbolized by a directed arc to the primary id of product.

4.3 The SCHEMA PREPARATION process

This phase is intended to prepare the conceptual interpretation by removing constructs that are no longer useful. Such is the case for the basic methods dedicated to managing and accessing the object instances. We just keep the application methods, i.e. those which implement user-oriented functions. In our schema, we can discard constructors (including the pseudo-constructor `add_detail`) and accessors. Two methods are kept, namely `customer::amount_due` and `order::cost_order`.

4.4 The SCHEMA DE-OPTIMIZATION processes

The size of the system being too small to exhibit realistic optimization constructs, we will concentrate on *untranslation* reasonnings. However, an important problem must be addressed in this process, namely *vicious* IS-A relations implementing *part-of* relationships (see Section 5.1).

4.5 The SCHEMA UNTRANSLATION process

We will consider three important untranslation rules that are intended to recover the original conceptual constructs of OO schemas.

A. Transforming the object-attributes

The schema includes two inverse object-attributes that are transformed globally into one-to-many rel-type `orders`.

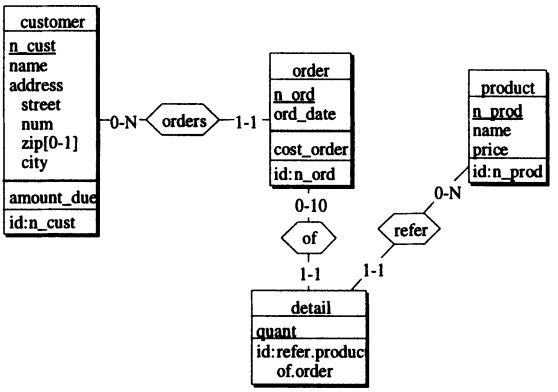

Figure 7 - The basic conceptual schema.

B. Transforming the complex multivalued attributes

Attribute `detail` is compound, multivalued, has an identifier and includes a foreign key. This is a typical implementation of a dependent (sometimes called *weak*) object class. This new class inherits the source name `detail` and the new rel-type is called `of`.

C. Transforming the foreign keys

Foreign key `n_prod` of newly defined class `detail` is replaced with a many-to-one rel-type called `refer` from `detail` to `product`.
The basic conceptual schema is presented in Figure 7.

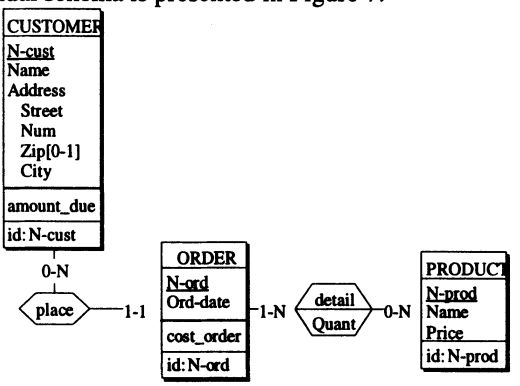

Figure 8 - The normalized conceptual schema.

4.6 The CONCEPTUAL NORMALIZATION process

The schema obtained so far is modified in such a way that it satisfies corporate methodological standards: model, naming conventions, graphical rules, etc. To illustrate the process, we propose in Figure 8 an equivalent schema, in which the names have been normalized according to local rules (e.g. class names in uppercase, attribute names capitalized, no underscores) and in which rel-types with attributes are allowed.

5. PROBLEM SOLVING IN OO REVERSE ENGINEERING

Despite its small size and its artificial nature, the program processed in Section 4 exhibits some of the most common problems that can occur when recovering the specifications of a structured collection of object classes. Though larger applications will include other kinds of problematic structures, the experiment above is sufficient to discuss the minimal requirements for a general methodology to perform OO application reverse engineering with success.

It is worth recalling some of the advantages of OO programs (even based on low level DMS, such as C++) as compared with more traditional development tools. Hierarchical object structures can be made explicit through recursively applying the tuple/set constructors. Explicit IS-A hierarchies can be explicitly declared. Methods provide a centralized support to integrity control and to logical relationships between objects (e.g. through inter-object navigation).

On the other hand, OO applications exhibit all the problems that have been found in classical legacy systems. The reason is three-fold. Firstly, even if it is state-of-the-art, each object manager has its own weaknesses that force the programmer to resort to traditional programming techniques to compensate for them. Secondly, many object managers lack features that are available in even the most primitive file managers (e.g. uniqueness constraint). Finally, and more important, the best programming environment, be it OO or not, cannot force programmers to work in the consistent and disciplined way the OO paradigm seems to naturally imply.

The rest of the section will discuss OO-specific problems as well as more general ones.

5.1 Ambiguity of the OO paradigm

One of the most disturbing observation is the fact that different scientific communities have given the concept of object class hierarchy distinct interpretations (Brachman, 1983). Two communities are concerned with the question we address in this paper, namely those from the programming and the IS/DB realms. Let us consider three object classes A, B and C; B and C being subclasses of A. On the one hand, OO programmers will generally consider that an object is in class A, or in class B or in class C, but in one of them only. For

instance, A can be declared an empty or *virtual* class if all the instances fall in either B or C. On the other hand, IS/DB people will consider that each B (or C) instance is an A instance as well. In short, in the programming realm, IS-A means **property inheritance** while in the IS/DB realm, IS-A means a **subset relation** between the populations. In more precise terms, if we call *prop(O)* the function that returns the properties (attributes, roles and constraints) of object class O and *inst(O)* the function that returns the current set of instances of class O, we have the following time-independent properties:

- according to the programming community:
  prop (A) $\subseteq$ prop (B) & inst (B) $\cap$ inst(A) = $\varnothing$
- according to the IS/DB community:
  inst (B) $\subseteq$ inst(A)
  hence prop (A) $\subseteq$ prop (B)

The semantics of the object classes in OO programs will be strongly influenced by the interpretation they have been based on. Recovering a conceptual schema from an OO program can involve an in-depth analysis of the intended meaning of IS-A relations, as illustrated in Figure 9 (left), which synthetizes one of the best illustrations of the problem.

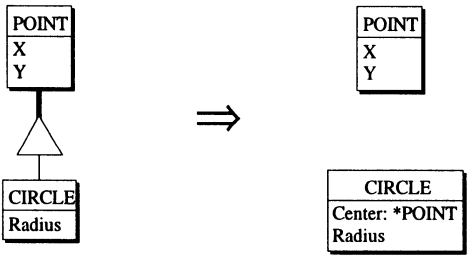

Figure 9 - Interpreting *vicious* IS-A hierarchies.

It has been provided by the Borland OO-Pascal (V7) documentation published some years ago. The very first example of class hierarchy presented in the tutorial is the following. Let us consider object class `point`, with attributes `X` and `Y`. We define a subclass `circle`, with which we associate a new attribute `radius`, and which inherits `X` and `Y` interpreted as the circle center coordinates. Generations of programmers were introduced to the OO approach with this particularly awkard example which suggests that circles form a special kind of points! The right side of Figure 9 proposed a more natural expression of the intended semantics.

5.2 Program analysis techniques

It is now quite obvious that program analysis can be the only way to extract the necessary knowledge on object behaviour to make hidden constraints and structures explicit. Visually analyzing the body of short object management methods can be sufficient in small applications. In can prove unreliable in large applications, in which objects tend to get more complex, and the methods much larger. Indeed, associating several hundreds of methods with each object class is not uncommon. In addition, in complex applications, not all the object behaviour will be localized in methods. Operations on sets of objects (sometimes called *object societies*) in which no member emerges as the *kernel* will often be wired in programs instead of in methods. Finally, the way the objects are manipulated in the program itself will often bring much interesting information on implicit constraints, or on the meaning of obscure attributes.

Hence the need for powerful program analysis techniques. We will briefly describe two classes of techniques, namely component dependency analysis and program slicing.

Two program data components (file, record, object, field, type, frame, constants) depends on each other if, at some point of the life of the program (compile time, run time) a property (name, type, structure, state, value) of one of them may depend of those of the other one. For instance, variables A and B can be considered mutually dependent if the program comprises either a statement assigning the value of A with B (or conversely) or a chain of statements resulting in A and B being assigned the same value at some point of program execution, or A being compared to B. The very meaning of such a dependency can vary according to the goal of the process: semantical similarity, same structure, evidence of a foreign key, etc. A special form of dependency graph is the dataflow graph in which directed arcs represents assignment operators only. This abstraction can be smaller and more precise that general dependency graphs (Anderson, 1996). Such techniques have been illustrated in Section 4 when trying to discover foreign keys.

In short, **program slicing** works as follows (Weiser, 1984). Let us consider a data object D (variable, constant, record type, file, etc.) of a program P, and any point p of P (statement, label, inter-statement point). Σ is the program slice of P with respect to criterion (p, D) if it is the subset of the statements of P such that, whatever the external conditions of execution, the state of D at point p is the same, whether Σ or P is executed. In other words, all the statements of P that can influence the state of D at p form the subprogram Σ. Normally, Σ is much shorter than P, and will be much easier to examine than P as far as understanding the behaviour of D is concerned. A typical example is the slice of a record type (D) at a file writing point (p). It is expected that this program excerpt includes all the statements that check and manage the records before writing them in the file. Further analyzing this slice will be easier than coping with the whole program. An

application of program slicing to database reverse engineering can be found in (Henrard, 1996).

For reasons that are out of the scope of this paper, several slices can be computed (Tip, 1994), (Horwitz, 1990). Optimistic slices include most of the statements of Σ as defined above, but some contributing statements may be lost. Conservative slices are garanteed to include all the statements of Σ, ... but some others as well. Of course, optimistic slices are shorter and cheaper to compute, but also less precise, than conservative ones.

5.3 Schema transformation

This is the basic tool that can be used to reliably derive schemas from source schemas. Schema transformations are ubiquitous techniques that have been proposed to support numerous processes in database engineering, and particularly in reverse engineering (Hainaut, 1993b). They are operators that replace constructs in a schema with other constructs. Generally, the second schema better meets definite criteria (normalization, minimality, compliance with a data model, etc.) the first one does not meet. The class of semantics-preserving transformations is of a particular importance. Such a transformation guarantees that the source and the final schemas convey the same semantics, i.e. they describe exactly the same application domain, and any situation that can be described by one of them can be described by the other one.

In reverse engineering, schema transformation will be mainly used in the Conceptualization phase. Indeed, replacing logical constructs with their conceptual equivalent, restructuring the schema to discard optimization constructs and normalizing conceptual schemas are typical schema transformations.

Due to the limited scope of this paper, we will only present three formal operators that are used in Section 4 (Figure 10 to 12). Being semantics-preserving, they can be read in both directions. The reader will find further information on these techniques in (Batini, 1992), (Hainaut, 1993b), (Hainaut, 1996), (Rosenthal, 1994) for example. In (Blaha, 1996), specific OO transformations are proposed.

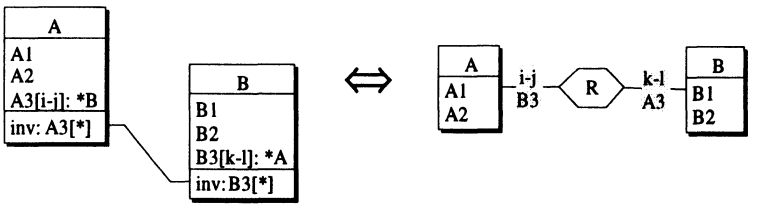

Figure 10 - Transformation of two inverse object-attributes into a rel-type (and conversely).

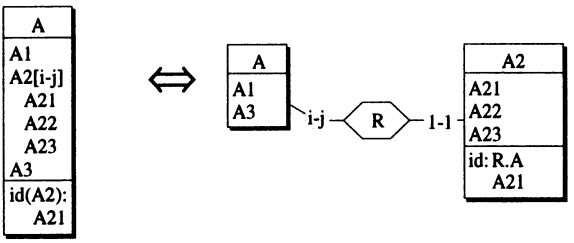

Figure 11 - Extracting a multivalued attribute as an autonomous object class (and conversely).

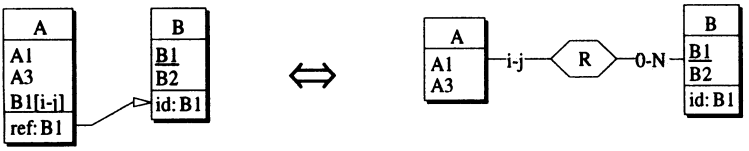

Figure 12 - Expressing a foreign key as a relationship type (and conversely).

6. DBRE CASE SUPPORT

From the analysis developed above, we can draw some minimal requirements that should be addressed by CASE tools intended to support reverse engineering of legacy systems, included OO applications. Besides natural functions related to specification entry, management and browsing, the tool must offer powerful program analysis processors and a large collection of transformation operators. From several large scale experiments, we learnt that the process is largely interactive and exploratory. Hence the need for extensibility to cope with unexpected situations.

To tackle data reverse engineering projects in a realistic way, we have developed a CASE tool suite which is being extended to the object paradigm, not only to recover the specifications of OO applications, but also to address the OO expression of traditional applications.

The DB-MAIN tool can be used either as a toolset to support system (forward and reverse) engineering, or as a CASE tool development environment (i.e. a meta-CASE tool).

In its current version (Version 3, November 1997), the tool offers a sophisticated support for forward and reverse engineering activities. More specifically, it includes the following functions and components:

- specifications management: access, browsing, creation, update, copy, analysis, memorizing;
- representation of all the project products, and of their relationships: schemas, views, source texts, reports, generated programs;
- view derivation and management;

- a generic, wide-spectrum, representation model for conceptual, logical and physical objects, according to the most popular value-based, entity-based and object-oriented paradigms;
- semantic and technical annotations attached to each specification object;
- multiple views of the specifications (4 hypertexts and 2 graphical views); some views are particularly intended for very large schemas;
- a toolbox of about 30 semantics-preserving transformational operators which provide a systematic way to carry out such activities as conceptual normalization, or the development of optimized logical and physical schemas from conceptual schemas, and conversely (i.e. reverse engineering);
- code generators; report generators;
- code parsers extracting physical schemas from SQL, COBOL, CODASYL, RPG and IMS source programs; O2 and C++ parsers are under development;
- interactive and programmable text analysers which can be used, a.o. to detect complex programming *clichés* in source texts, to build dataflow and dependency diagrams, and to compute *program slices;*
- a sophisticated name processor to clean, normalize, convert or translate the names of selected objects in a schema;
- a history manager which records the engineering activities of the analyst, and which makes their further replay and analysis possible;
- import and export of specifications;
- a series of *assistants.* An assistant is an expert module in a specific kind of tasks, or in a class of problems, intended to help the analyst in frequent, tedious or complex tasks. It allows the analyst to develop scripts which automate frequent processes. A library of predefined scripts is provided for the most frequent activities. Six assistants are available at present: global transformations, transformation script development, schema analysis, text analysis (including pattern searching, dependency analysis and program slicing), schema integration, foreign key analysis;
- meta functions that allow users to develop new specification objects* and new functions, particularly through the Voyager language.

Database reverse engineering cannot be carried out automatically. In addition, no unique tool can support every kind of DBRE projects, due to the large variety of problems that are encountered in practice, as opposed to development projects, which can profit from fairly standard approaches. Therefore, the DB-MAIN tool does not claim to solve DBRE problems automatically. On the contrary, it provides a rich collection of configurable toolboxes, together with rapid development tools to build ad hoc processors dedicated to specific projects.

The functions of the DB-MAIN CASE environment has been described in previous papers, such as (Hainaut, 1996d). Further information can be found at

* In the public version meta-objects and meta-relations cannot be created. Only meta-properties can be associated with builtin meta-objects.

http://www.info.fundp.ac.be/~dbm. A free education version of the tool is available. However, some of the OO extensions still are under development at present time (and therefore may be unstable), and will be made available on request only.

7. CONCLUSION

It is now clear that OO applications, beyond some positive aspects as far as program understanding is concerned, share many of the difficult problems that have been experienced in reverse engineering traditional, 3GL, legacy systems. The positive aspects derive from a (somewhat) more powerful data model and a stronger localisation of the code which manage the data objects. However, the weaknesses of some languages concerning data integrity (e.g. C++) and development practices inherited from old environments induce a complexity that is comparable to that of applications based on traditional languages. Therefore, it appears that the techniques developed can be adopted, with some adaptation, to tackle OO applications.

The experience also shows that data reverse engineering is far from an automated process, except for some specific processes, such as preliminary analysis of declarative code. Full human control is essential, even if it can be supported by powerful tools.

The last conclusion we would like to propose concerns the training of reverse engineering analysts (Hainaut, 1997). While forward engineering is fairly well understood and mastered, reverse engineering appears as an engineering discipline that makes use of complex theories and techniques known by very few professional only. For instance, the concept of program slicing, which is essential in program understanding, requires much effort both from the trainers and the trainees before being efficiently and reliably mastered.

8. REFERENCES

Andersson, M. (1994) Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag.

Anderson, M. (1996) Reverse Engineering of Legacy Systems: From Value-Based to Object-Based Models, Thèse n° 1521, EPFL Lausanne.

Baker, B. (1995) On Finding Duplication and Near-Duplication in Large Software Systems. *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press.

Batini, C., Ceri, S. and Navathe, S.B. (1992) Conceptual Database Design - An Entity-Relationship Approach, Benjamin/Cummings.

Blaha, M.R. and Premerlani, W.J. (1995) Observed Idiosyncracies of Relational Database designs, in *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press.

Blaha, M. and Premerlani, W.J. (1996) A Catalog of Object Model Transformations, *Proc. of the 3rd IEEE Working Conf. on Reverse Engineering*, Monterey, Nov. 1996, IEEE Computer Society Press.

Brachman, R.J. (1983) What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks, *IEEE Computer*, Oct. 1983.

Cattell, R., et al. (1994) The Object Database Standard: ODMG-93, Morgan Kaufmann.

Gall, H. And Klösh, R. (1995) Finding Objects in Procedural Programs. *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press.

Hainaut, J-L. (1989) A Generic Entity-Relationship Model, in *Proc. of the IFIP WG 8.1 Conf. on Information System Concepts: An In-depth Analysis*, North-Holland.

Hainaut, J-L., Chandelon, M., Tonneau, C. and Joris M. (1993) Contribution to a Theory of Database Reverse Engineering, in *Proc. of the IEEE Working Conf. on Reverse Engineering*, Baltimore, May, IEEE Computer Society Press.

Hainaut, J-L., Chandelon, M., Tonneau, C. and Joris M. (1993b) Transformational techniques for database reverse engineering, in *Proc. of the 12th Int. Conf. on ER Approach, Arlington-Dallas*, E/R Institute and Springer-Verlag, LNCS.

Hainaut, J-L., Englebert, V., Henrard, J., Hick J-M. and Roland, D. (1994) Evolution of Database Applications: The DB-MAIN Approach, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag.

Hainaut, J-L. (1996) Specification Preservation in Schema Transformations - Application to Semantics and Statistics, *Data & Knowledge Engineering*, **19**, 99-134, Elsevier.

Hainaut, J-L., Roland, D., Hick, J-M., Henrard, J. and Englebert, V. (1996b) Database design recovery, *in Proc. of CAiSE•96*, Springer-Verlag.

Hainaut, J-L., Henrard, J., Roland, D., Englebert, V. and Hick J-M., (1996c) Structure Elicitation in Database Reverse Engineering, *Proc. of the 3rd IEEE Working Conf. on Reverse Engineering*, Monterey, Nov. 1996, IEEE Computer Society Press.

Hainaut, J-L., Roland, D., Hick J-M., Henrard, J. and Englebert, V. (1996d) Database Reverse Engineering: From Requirements to CARE tools, *Journal of Automated Software Engineering*, **3**(1).

Hainaut, J-L. Hick, J-M., Henrard, J., Roland, D. and Englebert, V. (1997) Knowledge Transfer in Database Reverse Engineering - A supporting Case Study, in *Proc. of the 4th IEEE Working Conf. on Reverse Engineering*, Amsterdam, October 1997, IEEE Computer Society Press

Henrard, J., Hick, J-M., Roland, D., Englebert, V. and Hainaut, J-L. (1996) Techniques d'Analyse de Programmes pour la Rétro-Ingénierie de Base de Données, in *Actes de la conférence INFORSID'96*, Bordeau.

Horwitz, S., Reps, T. and Binkley, D. (1990) Interprocedural Slicing Using Dependence Draphs, ACM Trans. on Programming Languages and Systems **12**(1), Jan. 1990, 26-60.

Kung, C.H., Gao, J., Hsia, P., Lin, J. and Toyoshima, Y. (1993) Design Recovery for Software Testing of Object-Oriented Programs. *Proc. of the 1st IEEE Working Conf. on Reverse Engineering*, Baltimore, May 1993, IEEE Computer Society Press.

Newcomb, P. and Kotik, G. (1995) Reengineering Procedural into Object-Oriented Systems. *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press.

Penteado, R.D., Germano, F.S.R. and Masiero, P.C. (1996) An Overall Process Based on Fusion to Reverse Engineering Legacy Code, in *Proc. of the 3rd IEEE Working Conf. on Reverse Engineering*, Monterey, Nov. 1996, IEEE Computer Society Press.

Petit, J-M., Kouloumdjian, J., Bouliaut, J-F. and Toumani, F. (1994) Using Queries to Improve Database Reverse Engineering, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag.

Rosenthal, A. and Reiner, D. (1994) Tools and Transformations - Rigourous and Otherwise - for Practical Database Design, *ACM TODS*, **19**(2).

Rugaber, S., Stirewalt, K. and Wills, L. (1995) The Interleaving Problem in Program Understanding. *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July. 1995, IEEE Computer Society Press.

Sneed, H.M. and Nyáry. (1995) Extracting Object-Oriented Specification from Procedurally Oriented Programs. *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July. 1995, IEEE Computer Society Press.

Sneed, H.S. (1996) Object-Oriented COBOL Recycling. *Proc. of the 3rd IEEE Working Conf. on Reverse Engineering*, Monterey, Nov. 1996, IEEE Computer Society Press.

Sneed, H.S. (1996b) Encapsulating Legacy Software for Use in Client/Server Systems. *Proc. of the 3rd IEEE Working Conf. on Reverse Engineering*, Monterey, Nov. 1996, IEEE Computer Society Press.

Tip, F. (1994) A survey of program slicing techniques, Technical Report CS-R9438, CWI, (ftp://ftp..cwi.nl/pub/CWIreports/AP/CS-R9438.ps).

Vermeer, M.W.W. and Apers, P.M.G. (1995) Reverse Engineering of Relational Database Applications, in *Proc of Object-Oriented and Entity-Relationship Modeling*, Gold Coast, Australia.

Weiser, M. (1984). Program Slicing, *IEEE TSE*, **10**, 352-357.

Wieringa, R. (1997) *Advanced Object-Oriented Requirement Specification Methods*, Tutorial of the Int. Symposium of Requirements Engineering, Jan. 1997, Annapolis.

Yeh, A.S., Harris, D.R. and Reubenstein, H.B. (1995) Recovering Abstract Data Types and Object Instances from a Conventional Procedural Language. *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July. 1995, IEEE Computer Society Press.

BIOGRAPHY

Jean-Luc Hainaut is a professor in the Institut d'Informatique of the University of Namur (Belgium). His interests comprise database technology, database methodology, reverse engineering, system evolution, distributed component architecture and CASE technology. He is heading the multinational DB-MAIN D&D project, the purpose of which is to develop methods and tools to support complex and non-standard data engineering processes.

Vincent Englebert received his master thesis from the Institut d'Informatique, the University of Namur (Belgium), in 1992. He worked at the KUL (Leuven, Belgium) on the CLP(R) abstract interpretation. He is preparing a PhD in meta-modeling.

Jean-Marc Hick received his master thesis from the Institut d'Informatique, the University of Namur, Belgium, in 1991. Since 1993, he is working in the DB-MAIN team at the University of Namur. He is preparing a PhD on database application evolution and maintenance.

Jean Henrard received his master thesis from the Institut d'Informatique, the University of Namur, Belgium, in 1991. After some research in logic programming and telecommunication, he is currently preparing a PhD in database reverse engineering. Since 1993, he is working in the DB-MAIN team at the University of Namur. His primary research interests are database reverse engineering and program understanding.

Didier Roland received his master these from the Institut d'Informatique, University of Namur, Belgium, in 1992. Since 1993, he is working in the DB-MAIN team at the University of Namur. He is preparing a PhD on database application process modeling.

APPENDIX - THE C++ SOURCE CODE

```
1  #include <date.h>
2  #include <alloc.h>
3  #include <string.h>
4  #include <stdio.h>
5  #include <io.h>
6  class order;
7  class product;
8  class customer {
9  public :
10   typedef struct addr_ {
11     char street[40]; char num[5];
12     char zip[5]; char city[20];
13   } addrT;
14 private :
15   int n_cust; char name[30];
16   addrT address; order *first_ord;
17   customer *next;
18 public :
19   customer(int n_c, char *name,
       addrT adr);
20   customer *get_next();
21   int get_n_cust();
22   char *get_name();
23   addrT *get_address();
24   order *get_first_order();
25   void set_first_order(order *ord);
```

```
43   void add_detail(int n_p, int q);
44   order *get_next();
45   order *get_next_order();
46   int cost_order();
47 };
48 class product {
49 private :
50   int n_prod; char name[30];
51   int price; product *next;
52 public :
53   product(int n_p, char *n, int p);
54   int get_n_prod();
55   char *get_name();
56   int get_price();
57   product *get_next();
58 };
59
60 customer *find_customer(int n_c);
61 order *find_order(int n_o);
62 product *find_prod(int n_p);
63 customer *customers = NULL;
64 order *orders = NULL;
65 product *products = NULL;
66
67 customer *find_customer(int n_c)
68 { customer *cur;
69   cur = customers;
70   while(cur){
71     if(cur->get_n_cust() == n_c)
72       return(cur);
73     cur = cur->get_next();
74   }
75   return(NULL);
76 }
77
78 order *find_order(int n_o)
79 { order *cur;
80   cur = orders;
81   while(cur){
82     if(cur->get_n_ord() == n_o)
         return(cur);
83     cur = cur->get_next();
84   }
85   return(NULL);
86 }
87
88 product *find_prod(int n_p)
89 {  product *cur;
90   cur = products;
91   while(cur){
92     if(cur->get_n_prod() == n_p)
         return(cur);
93     cur->get_next();
94   }
95   return(NULL);
96 }
97
98 void read_not_null(char *str)
99 { str[0] = '\0';
100  while(str[0]=='\0') scanf("%s",str);
101}
102
103void read_not_null(int *val)
104{ *val = 0;
105  while(*val == 0) scanf("%d",
       val);
106}
107
108customer::customer(int n_c, char
       *n, addrT adr)
109{ if(find_customer(n_c))
       {delete(this);return;}
110  n_cust = n_c;  strcpy(name, n);
111  strcpy(address.street,
       adr.street);
112  strcpy(address.num, adr.num);
113  strcpy(address.zip, adr.zip);
114  strcpy(address.city, adr.city);
115  next = customers;  customers =
       this;
116  first_ord = NULL;
117}
118
119customer *customer::get_next()
120{ return (next);}
121
122int customer::get_n_cust()
123{ return(n_cust);}
124
125char *customer::get_name()
126{ return(name);}
127
128customer::addrT
       *customer::get_address()
129{ return(&address);}
130
131order *customer::get_first_order()
132{ return(first_ord);}
133
134void customer::set_first_order
       (order *f)
135{ first_ord = f;}
136
137int customer::amount_due()
138{ int total;  order *cur;
139  total = 0;
140  cur = get_first_order();
141  while(cur){
142    total = total +
               cur->cost_order();
143    cur = cur->get_next_order();
144  }
145  return(total);
146}
147
148order::order(int n_o,customer *cus)
149{ int i;
150  order *cur, *prev;
151  if(find_order(n_o)){
152    delete(this);
153    return;
154  }
155  n_ord = n_o;
156  for(i=0; i <10; i++){
157    detail[i].n_prod = 0;
158    detail[i].quant = 0;
159  }
160  cust = cus;  next = orders;
161  orders = this;
162  cur = cust->get_first_order();
163  prev = NULL;
164  while(cur){
165      prev = cur;
166      cur = cur->get_next_order();
167    }
168  next_of_cust = NULL;
169  if (prev)
       prev->next_of_cust = this;
170  else cust->set_first_order(this);
171}
```

```
172
173 customer *order::get_customer()
174 { return(cust);}
175
176 int order::get_n_ord()
177 { return(n_ord);}
178
179 TDate *order::get_date()
180 { return(&ord_date);}
181
182 order::detT *order::get_detail(int i)
183 { return(&detail[i]);}
184
185 void order::add_detail(int n_p, int q)
186 { int i;
187   if(!find_prod(n_p))
188     printf("the product does not
       exist\n");
189   i = 0;
190   while(i<10){
191     if(detail[i].n_prod == 0)
          break;
192     if(detail[i].n_prod == n_p)
          break;
193     i++;
194   }
195   if(i<10)
196     if(detail[i].n_prod != n_p){
197       detail[i].n_prod = n_p;
198       detail[i].quant = q;
199     }
200 }
201
202 order *order::get_next()
203 { return(next);}
204
205
206 order *order::get_next_order()
207 { return(next_of_cust);}
208
209 int order::cost_order()
210 { int total, i; product *prod;
211   total = 0;
212   for(i=0; i<10; i++){
213     if(detail[i].n_prod == 0)
          break;
214   prod=find_prod(detail[i].n_prod);
215     total = total + detail[i].quant
216                   * prod->get_price();
217   }
218   return(total);
219 }
220
221 product::product(int n_p, char *n,
       int p)
222 { if(find_prod(n_p)){
223       delete(this);
224       return;
225     }
226   n_prod = n_p; strcpy(name, n);
227   price = p;
228   next = products; products = this;
229 }
230
231 product *product::get_next()
232 { return(next);}
233
234 int product::get_n_prod()
235 { return(n_prod);}
236
237 char *product::get_name()
238 { return(name);}
239
240 int product::get_price()
241 { return(price);}
242
243
244 void new_cus()
245 { int n_cust;
246   customer::addrT addr;
247   char name[30];
248   printf("new customer : \nCustomer
        code");
249   read_not_null(&n_cust);
250   printf("Customer's name : ");
251   read_not_null(name);
252   printf("address of
        customer\nstreet : ");
253   read_not_null(addr.street);
254   printf("number : ");
255   read_not_null(addr.num);
256   printf("zip code : ");
257   scanf("%s", addr.zip);
258   printf("city : ");
259   read_not_null(addr.city);
260   if(!new customer(n_cust, name,
        addr))
261     printf("err: cust not created\n");
262 }
263
264 void list_cus()
265 { customer *cur;
266   cur = customers;
267   while (cur){
268     printf("%s  amount due : %d\n",
269            cur->get_name(),
               cur->amount_due());
270     cur = cur->get_next();
271   }
272 }
273
274 void new_stk()
275 { int n_prod,price; char name[30];
276   printf("new product\n product
        number:");
277   read_not_null(&n_prod);
278   printf("name : ");
279   read_not_null(name);
280   printf("price : ");
281   read_not_null(&price);
282   if(!new product(n_prod, name,
        price))
283     printf("error : product
        exists\n");
284 }
285
286 void list_stk()
287 { product *cur;
288   cur = products;
289   while(cur){
290     printf("%s->%d\n",
            cur->get_name(),
            cur->get_price());
291     cur = cur->get_next();
292   }
293 }
294
295 void new_ord()
296 { int n_ord, n_cus,n_prod, quant;
297   customer *cust;order *ord;
```

```
298
299   printf("New order\norder num: ");
300   read_not_null(&n_ord);
301   cust = NULL;
302   while(!cust){
303     printf("customer number : ");
304     read_not_null(&n_cus);
305     cust = find_customer(n_cus);
306   }
307   ord = new order(n_ord, cust);
308   if(!ord){
309     printf("err: order exists\n");
310     return;
311     }
312   printf("product num (0=end): ");
313   scanf("%d", &n_prod);
314   while(n_prod != 0){
315     printf("quantity: ");
316     read_not_null(&quant);
317     ord->add_detail(n_prod, quant);
318     printf("product n° (0=end): ");
319     read_not_null(&n_prod);
320   }
321 }
322
323 void list_ord()
324 { order *cur; order::detT *det;
325   int i;
326   cur = orders;
327   while(cur){
328     printf("order : %d\n",
              cur->get_n_ord());
329     for(i=0; i<10; i++){
330       det = cur->get_detail(i);
331       printf("\t%d  %d\n",
332          det->n_prod, det->quant);
333     }
334     printf("\tcost order : %d\n",
335          cur->cost_order());
336     cur = cur->get_next();
337   }
338 }
339
340 int main()
341 { char choice;
342   choice = ' ';
343   while(choice != '0'){
344     printf("1 New customer\n2 New
       stock\n3 New order\n4 List of
       customers\n5 List of stocks\n6
       List of orders\n0 end\n");
345     scanf("%c", &choice);
346     switch(choice){
347       case '1' : new_cus();
348         break;
349       case '2' : new_stk();
350         break;
351       case '3' : new_ord();
352         break;
353       case '4' : list_cus();
354         break;
355       case '5' : list_stk();
356         break;
357       case '6' : list_ord();
358         break;
359       }
360     }
361 }
```