Object-Oriented Reengineering of COBOL Applications

A. Fantechi and P. Nesi
Department of Systems and Informatics
Faculty of Engineering, University of Florence
Via S. Marta 3, I 50139 Florence, Italy
fantechi@dsi.ing.unifi.it, nesi@ingfi1.ing.unifi.it

August 22, 1997

Abstract

The object-oriented paradigm is presently considered as the approach which best guarantees investments for renewal. It allows to produce software with a high degree of reusability and maintainability, thus satisfying to a certain extent also quality characteristics. In this perspective, several methods for reengineering old applications according to the object-oriented paradigm were defined and proposed. Anyhow, the simple adoption of an object-oriented language, or the encapsulation of the old application within an object-oriented interface, does lead to earn the high software quality achievable with a complete object-oriented design trajectory. Rather, a process of object-oriented reanalysis of the application is needed. In this paper, a method and tool (C_2O^2, COBOL to Object-Oriented) for analysing COBOL applications in order to extract an object-oriented analysis is presented. The tool identifies classes and their relationships by means of a process of understanding and refining in which COBOL data structures are analysed, converted in classes, aggregated, and simplified semi-automatically. The analysis process also helps in detecting data structures which can cause failures passing to the next millennium, and in this respect solutions are suggested.

1 Introduction

According to the trend of the information technology, software languages are evolving towards new paradigms. As a consequence, applications written in the past need to be maintained aligned with the state of the art. This process is often highly expensive since the language evolution proceeds in high steps – e.g., from procedural to object-oriented, from textual to visual, from operational to descriptive, etc., approaches [1]. The literature about software renewal is quite extensive – e.g., [2], [3], [4], [5], [6]. A solution for minimising the costs of renewal was proposed by the so-called legacy-techniques. These suggest to reuse old applications by encapsulating them with suitable interfaces. Unfortunately, these techniques are not suitable when the system software must be maintained and updated by modifying the internal behaviour during its maintenance. This is what usually happens in the information systems which are used in banks, public administrations, and other large organisations. Most of these applications have been traditionally written in COBOL and now there is a strong interest in porting them on new languages and platforms and updating them to the current quality standards. This is a huge task (more than 180 billions lines of COBOL code are currently running all over the world) which deserves to be tackled with advanced reengineering methodologies and tools.

Presently, the most important requirements for the renovation of COBOL applications are (i) the updating of the user interface towards a windowing system, (ii) the addition of new functionalities, and (iii) the migration from a classical client/server architecture to a fully distributed architecture based on the new intranet features. For satisfying the first feature at low cost, the market offers several screen generators under MS-Windows for COBOL. For the second feature, a deep knowledge of the system is needed. This has to be regarded as a full maintenance process of the old code. The third feature is hard to be satisfied without reengineering the whole application [6]. This is due to the fact that COBOL applications are typically based on transactions performed on a centralised database. For these reasons, a strong process of reengineering is frequently mandatory and cannot be solved by a mere translation of COBOL programs into a newer language. The object-oriented paradigm (OOP) allows software to be produced with high degrees of reusability and maintainability, thus satisfying to a certain extent also quality characteristics [7], [8], [9], [10], [11], [12]. Hence, the OOP is presently considered as the one which best guarantees the investment for renewal [11]. Presently, the most promising and used object-oriented languages are C++, Java and recently also Object COBOL [13].

The above mentioned benefits of the OOP are not obviously automatically guaranteed by the simple adoption of an object-oriented programming language, though this is a common opinion. The compilation in C++ of an application written in C does not transform it in an object-oriented application (the same statement holds for COBOL and Object COBOL, Pascal and Object Oriented Pascal, etc.).

In the literature, several methods for reengineering procedural applications according to the OOP were defined and proposed – e.g., [14], [15], [5], [6]. In order to pass from a procedural version of a program/application to an object-oriented version, a process of reanalysis is mandatory. The focus must pass from data transformations (i.e., procedures) to data structure, categories and entities (i.e., classes and objects). This process leads to the identification of the classes which model the application domain and their organisation [7], [16]. If the problem domain analysis is not performed, the reengineering leads to produce a large number of classes, without aggregating them and, thus, without exploiting the capability of the OOP. The advantage of performing an object-oriented analysis of the system is particularly evident in the case of motivation (iii) above, that is the migration of centralised database COBOL applications to fully distributed architectures.

A large effort has been accomplished for defining algorithms to migrate, transform, reengineering procedural programs into object-oriented ones – e.g., [17], [15]. On the other hand, the reengineering of COBOL applications towards object-oriented is a harder problem, with respect to the reengineering of high-level procedural languages such as C, FORTRAN, Pascal, since the semantic gap between COBOL and object-oriented languages is much more relevant. In the literature, some tools and algorithms for converting COBOL applications into object-oriented ones have been presented – e.g., REDO [18], REDOC [19], [6], MOORE [20], CORECT (`http://www.acm.co.uk`).

In this paper, a method/tool (C_2O^2, COBOL to Object-Oriented) for analysing COBOL applications (collections of programs written in COBOL for covering a whole applicative domain) in order to extract its object-oriented analysis is presented. This tool identifies classes and their relationships by means of a process of understanding and refining in which COBOL data structures are analysed, aggregated, and semi-automatically simplified. The result is a complete object-oriented analysis of the application domain. The analysis is smart enough to associate structurally similar data having different names as usually happens in COBOL programs. The resulting analysis is largely independent of the final object-oriented language which

can be chosen to code (or to code again) the application.

This paper is structured as follows: Section 2 describes the overall transformation process from COBOL programs to object-oriented ones. Section 3 details the abstraction algorithm used for the analysis of the application. Section 4 shows how the analysis process supported by the C_2O^2 tools also helps in detecting data structures which can cause problems passing to the next millennium; and in this respect solutions are suggested. Section 5 describes the prototype C_2O^2 tool that implements the abstraction algorithm. A discussion on related works is included in the concluding Section 6.

2 From COBOL to Object-Oriented

A COBOL application usually consists in a collection of modules placed in distinct files sharing several data structures and obviously the application domain. The typical COBOL application is organised around a kernel made up by one or more menu modules and by a collection of independent modules which perform the logically separated operations. The organisation of modules inside a COBOL application generally reflects the functional decomposition process usually adopted in COBOL to carry out the system design. In COBOL these is a neat separation between modules, since each module can be seen as a self-standing one with its own well-defined functionality, different from all the others. This division can simplify the transformation work since the scope of the code is limited.

COBOL programs can be classified in three fundamental types: SUBprograms, BATCH programs and ONLINE programs. BATCH and ONLINE modules are typically indicated as main programs. For each type of module it is necessary to adopt different conversion strategies which will be clearly specified in the following.

SUBprograms are modules called to perform functions which are too complex or heavy to be included in the main program, or which are used to implement generally used functions or utilities required by several other programs. Roughly, the goal of SUBprograms is to modify the value of some parameters according to some others, or to access archives, print reports, and presentations. SUBprograms can be regarded as procedures or methods according to the OOP, for transforming data/objects, for printing data/objects, for loading/saving data/objects, etc.

BATCH programs have as their goal the access to files and/or databases to produce reports and synthetic presentations of the databases. The files, databases, reports and views are the objects of program action, though differently from the case of reports or views which structure its control flow; in the case of operations on files or databases, the control flow of the program is established by the operation itself (cancellation, insertion or modification of data). These functionalities can be regarded as special methods of the main classes of the databases or of the classes which are closer to the database. Typically, batch transactions are performed on categories of objects. Thus, the identification of classes can be profitably performed on these modules.

ONLINE programs process on line transactions. They are built around the transaction they are processing. The structure of such transactions is reflected by the forms of the interface prepared to call the transaction. An ONLINE transaction can access many files or databases from a single form. Therefore, there is a one-to-many relationship between the source and the targets of the action, each of which can be seen as an object.

A COBOL program is always structured in four parts (*divisions*):

- IDENTIFICATION DIVISION: reporting the title and author of the program and other information needed to identify the module;
- ENVIRONMENT DIVISION: reporting the configuration sections of the host system, and the definition sections for the file or I/O device control;
- DATA DIVISION: reporting the physical organisation of data on both I/O devices and central memory. It presents a WORKING-STORAGE section and a LINKAGE section;
- PROCEDURE DIVISION: reporting the sequence of operations to be performed on the defined data, such as the reading/writing access operations on external devices, numerical computation, controls and alternatives.

A central role in the transformation from COBOL applications into an object-oriented design is performed by the analysis of DATA DIVISION, which actually contains all the information needed to create a representation of the data structures of the application domain under examination – i.e., the definitions of data structures in general, structure of files and records in particular.

2.1 The Transformation Process

The preliminary step of the transformation process is the classification of modules as BATCH, SUB or ONLINE. The transformation of a COBOL SUBprogram is a specific case of the transformation of a main program, since it requires to recognise its specific functionality in the context of the modules which activate it. For these reasons, we start with the discussion of the transformation for main program modules, that is BATCH and ONLINE in this order. Typically, the BATCH modules of an application contain the great part of the data structures. In other modules, these are redefined with minor changes for specific purposes.

As depicted in Fig.1, the transformation process for a COBOL application is iterative and can be performed in several phases. In each phase, the Section is reported in which it is discussed. The phases are:

1. **Identification of COBOL data structures** – The first step of the transformation of a COBOL application (constituted by ONLINE, BATCH and SUB modules) is the analysis of the code in order to identify all labels and strings which specify data and data structures. This requires the analysis of all COBOL data structures of the application domain. The names of structures and variables (labels) are collected in a local database with their relationships and formats. Please note that, in COBOL, equivalent formats (as to memory allocation space and variable types – e.g., numeric, alphanumeric) can be defined in several syntactically different ways. This increases the complexity of the mechanisms for collecting and identifying types.
2. **Elimination of redundant definitions of formats (unification of formats)** – In the second step, a set of criteria is applied in order to identify and eliminate redundant (i.e., duplicated) formats. This is initially performed by identifying the essential formats used in the whole application, then by applying a set of criteria for identifying equivalences among instance variables with compatible formats (unification of formats). The identification of a minimal number of data types is mandatory for reducing the complexity of the system and producing a real analysis of the system itself. This phase generates the collection of the essential formats used in the whole application (a list of equivalencies is maintained).

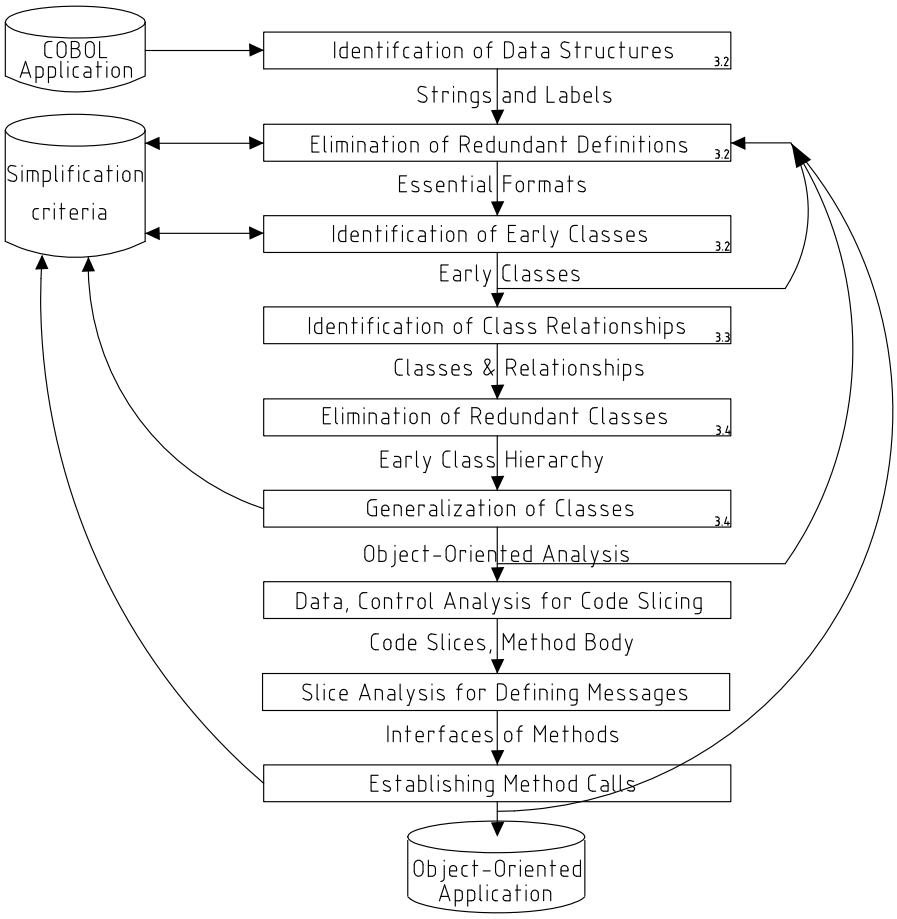

Figure 1: *The process for reengineering COBOL applications towards object-oriented.*

3. **Identification of COBOL data structures as early classes** – The third step presents the identification of classes on the basis of the data structures and variables. According to the previous phase, variables are classified on the basis of their type (internal structure in terms of other variables or by using formats) and, then, classes are assigned to them. Please note that in COBOL applications many data structures and variables are usually repeated in the modules – e.g., adding a prefix or a postfix; the names of these variables must be considered as synonyms. In this phase, on the basis of an incrementable table of synonyms, several synonyms of instance variables are eliminated (maintaining also in this case a list of equivalencies) (unification of labels). The table of synonyms can be defined by the user or is automatically produced by activating a set of criteria. The process of elimination of synonyms is called "label unification" and leads to reduce the number of simple classes (objects) modelling the atomic data types. Once the minimal data structures are identified, they can be considered as the elementary categories of the application domain, thus early prototypes of classes. If the reduction of formats is not satisfactory, the process can restart from the second phase, by adding/changing criteria for improving the processes of unification.

4. **Identification of early class relationships** – In the forth step, structured variables are defined as composite classes and, thus, according to the OOP the analysis process continues by establishing relationships of aggregation, association and specialisation (i.e., is-part-of, referred-by). In this step, the variables defined by using the COBOL keyword REDEFINES lead to specialised classes. This is very useful for the further association of code to classes.

5. **Elimination of redundant classes (unification of classes)** – The fifth step is devoted to the elimination of redundant classes. This is performed by identifying among the obtained classes those which have the same data structure in terms of instance variables (equivalent attributes). In this case, the classes are unified and the synonyms maintained.

6. **Generalisation of classes** – In the sixth step, generalised classes are identified. This is performed by identifying among the obtained classes those which can play the role of subclasses of a unique superclass. These are identified by verifying the presence of an intersection among the data structures of the candidate subclasses. Thus, also in this case, relationships of specialization are defined. The result of this step is the object-oriented reanalysis of the application domain. The output of this phase can be unsatisfactory due to a higher number of classes and/or a poor class organisation; thus, a further iteration could be needed by restarting from the second step. An additional iteration can also be required if new criteria for simplifying formats and classes are defined on the basis of the experience acquired by observing the program transformation while performing the previous steps.

7. **Analysis of data and controls for code slicing and allocating slices as method body** – The seventh phase consists in analysing the code in order to identify the accesses to data (e.g., MOVE, SELECT, UPDATE, INSERT, DELETE, READ and WRITE operations). With this operation, it is possible to evaluate the relationships between classes and the code of modules. To this end, the data bases of equivalencies are used and the number of accesses are counted to identify the most probable class for each given slice of code. Each code slice is the candidate body for a method of a class among those which are in the module. Their arrangements in the class hierarchy depends on the position

which has been assigned to the attributes (variables) that slices adopt. Equivalent pieces of code are identified and, thus, methods for superclasses are created avoiding code duplication.

8. **Analysis of slices for defining parameters and local variables** – The eighth step is devoted to identifying parameters and local variables of methods (the above mentioned slices of code). In fact, once a slice is reallocated as a method, a part of references to variables contained in the slice is resolved by updating the old names with the new class attribute names (according to the above generated list of synonyms). References to other variables which are out of the class scope must be regarded as local variables or method parameters. The former are those which are only locally defined and are not used in other modules, while the latter are the remaining ones.

9. **Establishing calls among methods of classes** – Step nine consists in establishing the method calls among methods of different classes in order to close the transformation process by eliminating all pending unresolved references. In this phase, some non-optimal solutions may be detected, for example due to the presence of non-simplified classes or class hierarchy, duplicated code, etc. For these reasons, the process can be reiterated by starting from the second step.

In order to identify the best criteria to be applied some early analysis can be performed on a restricted number of modules. Then, in order to identify the most relevant classes, to start from the BATCH modules is suggested. Once the main classes with their relationships are found, the other modules can be added.

In the next Section, the first six phases are described, since they are crucial for the object-oriented analysis of the application. The last three phases can be in practice performed when classes and their relationships are identified.

3 Analysis of the Application Domain

As described in the previous, section the analysis of the data structures is central in the identification of the object-oriented analysis of the application domain. As discussed in Section 6, it is also the point in which most of the methods presented in the literature and of the commercial tools for reengineering of COBOL application are less effective.

3.1 COBOL Data Structures

The identification of redundant data structures in COBOL applications is not at all an easy task; actually, COBOL is extremely clear in the definition of the procedural part, but it is quite obscure in the parts regarding data definition and configuration, since sometimes it allows several interpretations and some others equivalent structures can be defined in several different ways. The motivation for a non-immediate identification of redundancies is due to different causes briefly schematised in the following:

- Since a COBOL programmer is free to define at his/her will the data structure in the archives, it is not rare that different programmers, in different modules of the same application, give different (maybe similar, but not equal) names to the same data and data structures. This is an example, taken from a real application:

```
05 W-DATE
   10 DD   PIC 99
   10 MM   PIC 99
   10 YY   PIC 99
..

05 W-DT
   10 YEAR  PIC 99
   10 MONTH PIC 99
   10 DAY   PIC 99
..

05 W-D
   10 D PIC 99
   10 M PIC 99
   10 Y PIC 99
..
```

- Since the visibility of identifiers is global, even for a single programmer there is the need to give different names to the same structure of a FILE, if it occurs two times in the definition of data in a module. Again, an example taken from the same application:

```
01 FINV
   02 FINV-DATE
      03 FINV-DATE-YY
      03 FINV-DATE-MM
      03 FINV-DATE-DD
.....

01 FBID
   02 FBID-DATE
      03 FBID-DATE-YY
      03 FBID-DATE-MM
      03 FBID-DATE-DD
.....
```

- In the definition of the elementary data, the specification of the same format of the PICTURE clause in different ways is possible; this is often used to design logically different data, with the same physical memory occupation:

```
01 W-YEAR PIC 99
..
```

```
01 W-YEAR PIC 9(2)
..

01 W-YEAR PIC 9(002)
..
```

- Having a complete control on the physical representation of data in memory, it is also possible that logically similar data structures are defined as elementary in a module and as compound in others. An example can be given by the representation of the date, for which no international standard is given, but rather a plethora of national or even regional uses of representing a date are applied: this can bring (even in the same program) to different structures used for representing a date, according to the local use of the date itself.

```
01 W-DATE
   05 W-YY  PIC 99
   05 W-MM  PIC 99
   05 W-DD  PIC 99
....

01 W-DATE-N PIC 9(6)
....
```

3.2 Reduction of the Simple Types

In the first phase, the names of variables and data structures are identified. In the second and third phases, a set of criteria for detecting and automatically solving the above mentioned problems is applied. Thus, it is possible to unify the different names which are used for simple data and data structures in different COBOL modules. This process is performed by establishing a list of synonyms for both formats and labels (names of variables and structures). This mechanism strongly reduces the main causes of redundancy, and thus the number of classes. An example regarding the first reason for redundancy, namely the use of different names by different programmers, is the equivalence: `YEAR = YY = Y`. The static dictionary can answer to the problems of global synonyms. Local synonym problems are more common, due to a widely used convention to make identifiers more meaningful.

It is possible to specify other criteria for finding synonyms, for example: (i) discard numeric suffixes, like in: `YEAR-000`; (ii) Hungarian notation: `EV-IN-DT-YEAR` (Event/Initial/Date/Year). The simultaneous use of these filters slows down the conversion, and usually does not provide better results. Actually, it is likely that a COBOL application uses a single convention for the identifiers. A quick inspection of the source code is enough to recognise the used convention and to select the most useful filter accordingly. Please note that most of COBOL applications have been maintained by different people in different times and situations, hence finding uniformity in the code is not justified. It is also be possible to attempt a further refinement of the equivalence criteria, in order to enlarge the automatic recognition cases.

The evolution and improvement of the equivalence criteria are strongly needed when a large application is analysed since different styles and techniques are usually existing in the modules of the application.

3.3 From COBOL Data Structures to Classes

The early identification of classes and their relationships is performed by using of the following subphases:

- identification of composite classes, relationships of is-part-of and is-referred-by;
- identification of specialisation for REDEFINES (is-a).

The identification is performed through the analysis of the definition of the elementary data at the field level (see the PICTURE construct) which, in particular, specifies the visualisation and conversion formats. This parameterises classes, and is used to calculate the memory occupation of numeric or alphanumeric data and to define the default values. For example, the following clauses specify elementary data:

```
10 fiscal-code  PICTURE  AAABAAAB99A99BA999X
10 YEAR         PIC  99
10 INCOME       PICTURE  S9(7)P
10 CARTOON      PICTURE  A(7)BA(5) VALUE "GOOFY"
```

The first definition establishes that the `fiscal-code` (i.e., social security number) field is allocated in 16 memory locations of one byte each and is formatted as the example: `GMM MNL 68R10 A509W` (a typical Italian fiscal code).

The identification of the compound classes is performed through the analysis of the layout of the DATA DIVISION, WORKING-STORAGE SECTION and LINKAGE SECTION structures of the main program. The simpler procedure for the conversion of compound data is the transformation of each record in a class and of each field in an instance variable (class member), whose class is specified by the corresponding elementary class if the field is elementary, or by another class, if the record contains a further record in its turn.

The identification of the classes is performed through an iterative strategy based on Automatic Abstraction Algorithm for recognising the abstractions and simplifying the redundancies. The application of the algorithm provides the automatic recognition of the structures which form the program, by means of the production of tables and diagrams which allow the user to recognise key aspects of the design or possible areas of refinement of the analysis process. It is then possible to use this information to enrich the basic knowledge on the application and to reapply, iteratively, the analysis algorithm. All the information related to names of variables are collected in a knowledge base. In the following, it is shown, by means of a particularly simple example, how the Automatic Abstraction Algorithm works on its tables and diagrams.

The first operation performed is the substitution of the synonyms for the base names. In the following, a typical example of COBOL data structure definition is reported:

Module ONE

```
01 PATIENT
   05 NAME      PICTURE X(40)
```

```
   05 BIRTH
      10 DD   PIC 99
      10 MM   PIC 99
      10 YY   PIC 99
   05 BIRTH-N REDEFINES BIRTH PIC 9(6)
   ...
...
```

Module TWO

```
01 CLIENT
   02 ID  PICTURE XXB999
   02 LAST-ORDER
      03 YEAR  PICTURE 99
      03 MONTH PICTURE 99
      03 DAY   PICTURE 99
   02 NAME     PIC X(40)
   ...
...
```

In the example, it is possible to note how labels `YY`, `MM`, and `DD` in *Module ONE* are equivalent to `YEAR`, `MONTH` and `DAY` in *Module TWO*. This consideration can be performed simply by observing the code, and so it can be introduced in the knowledge base about the application before the analysis (as we suppose in this case). Alternatively, it can be recorded in the knowledge base when the abstraction process has already produced the first diagrams and results. As will be shown in the following, once two names are recognised to be synonyms, the synonym disappears since it is completely equivalent to the first name.

The next operation is the building of tables containing important information recovered during the analysis of the application; in particular, the following tables are built:

- Table of system Modules (TM);
- Table of Types (TT): table of formats;
- Table of the Instance Variables (TVI): table of elementary classes, see Fig.2;
- Table of the Classes (TC).

Each module/file of the COBOL application under analysis is stored in TM; in our case TM has the following content:

Table of system Modules, TM		
Module_ID	name	path
1	ONE	/WORK/CO2OC/SBN/
2	TWO	/WORK/CO2OC/SBN/
..	....	...

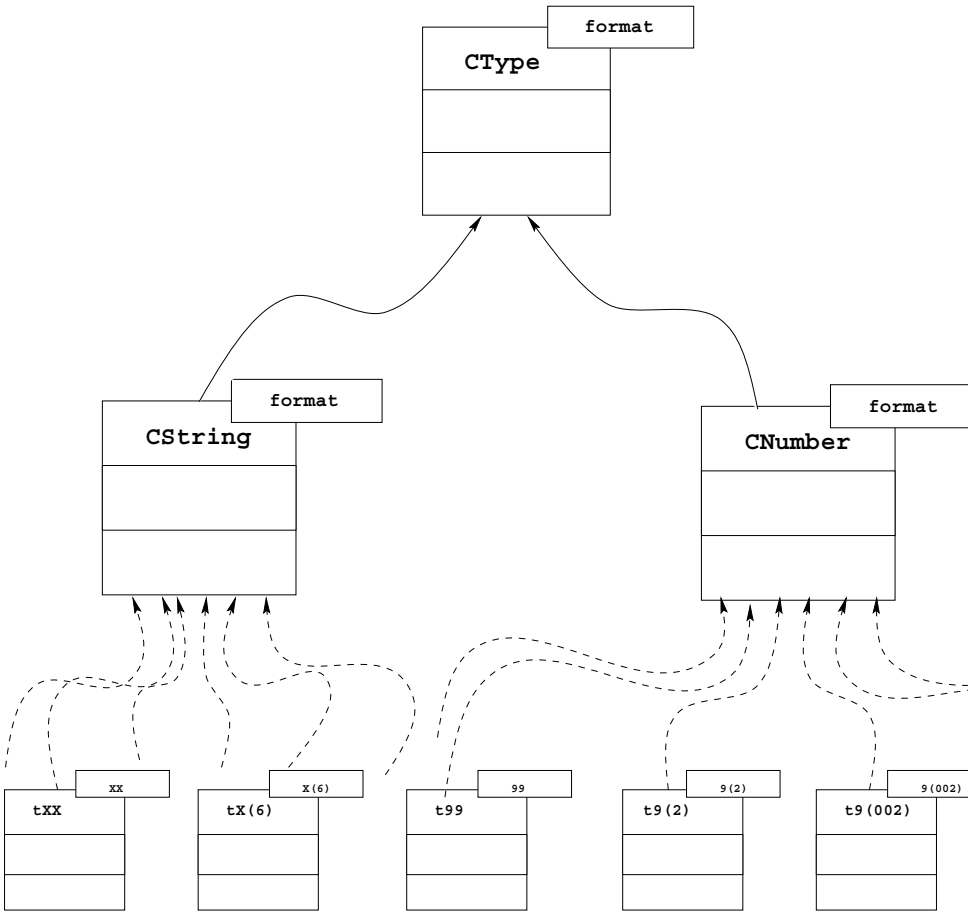

Figure 2: *Relationships among fundamental classes and those which are specifically defined for the application: dashed lines represent in this case the process of class instantiation with parameters; continuous lines are is-a relationships.*

The only data types which can be directly defined in COBOL are those reported in the following table. Each type is listed with an example of its use and the class that has been defined as fundamental for any object-oriented application:

Type	example	Library Classes
Numeric	S999V999	CNumber<"S999V999">
Alpha	AABBAAA	CString<"AABBAAA">
Alphanum	A(6)99A999X	CString<"A(6)99A999X">

CNumber<format> and CString<format> are subclasses of CType<format> class, all these classes are provided in a library. These have been parameterised on their format, as shown in Fig.2. In practice, each elementary class defined in the process can be considered as a specialisation of parameterised class CType<format> provided with the tool (see for example classes tXX, tX(6), etc., of Fig.2). This mechanism is implemented in the tool by instantiating an object of classes CNumber or CString according to their nature and the above table.

Each new format generates an entry in the Table of Types, TT, where A means alphanumeric and N numeric; in the last column the dimension in bytes is reported:

Table of Types, TT			
Format_ID	format	Type	Dimension
1	X(40)	A	40
2	99	N	2
3	9(6)	N	6
4	XXB999	A	5
...	...	...	...

Each elementary field label of COBOL data structures generates an entry in table, TVI. Homonymous labels generate different entries, see for example for NAME:

Table of the Instance Variables, TVI			
TVI_ID	LABEL	Class_ID	Format_ID
1	NAME	1	1
2	DAY	2	2
3	MONTH	2	2
4	YEAR	2	2
5	BIRTH-N	6	3
6	ID	3	4
7	YEAR	4	2
8	MONTH	4	2
9	DAY	4	2
10	NAME	3	1
...	...	..	..

The last column, Format_ID, records the references to the formats as indexes into TT. In this table, we can also note that labels YY, MM and DD have disappeared, begin replaced by their synonyms, as we have seen before. Column Class_ID reports the references to the compound classes considered in the analysis of the definitions of records in the DATA DIVISION and, thus, in TC itself, according to an is-part-of relationship.

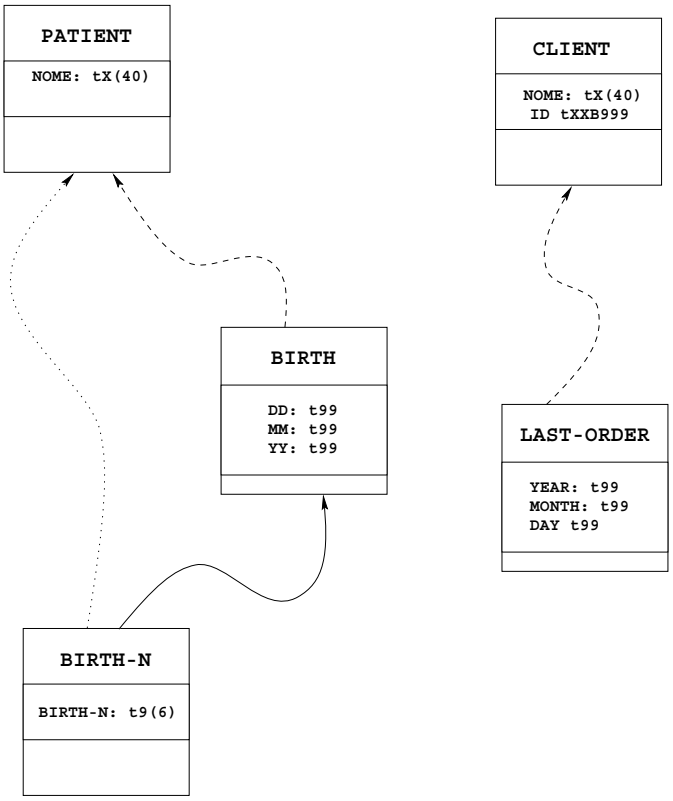

Figure 3: *Relationships among classes: is-a as continuous lines, is-part-of as dashed lines, is-referred-by as dotted lines.*

Each record of the COBOL data structures generates an entry in the following table, TC. The relationships automatically obtained after the first phase are reported in Fig.3. The following TC table reports classes and their relationships according to the OOP [7], [16]. Each entry of columns is-a, is-part-of and is-referred-by reports the Class_ID representing class relationships. In effect, each entry is a list of Class_IDs which in this case contains only an element. In the example reported, only single relationships have been found. In TC the final dimensions in bytes of class instances are also reported.

Table of the Classes, TC							
Class_ID	class name	Module_ID	is-part-of	is-a	is-referred-by	dim.	cost
1	PATIENT	{1}	-	-	-	46	46
2	BIRTH	{1}	{1}	-	-	6	6
3	CLIENT	{2}	-	-	-	51	51
4	LAST-ORDER	{2}	{3}	-	-	6	6
5	BIRTH-N	{1}	-	{2}	{1}	6	0
..	..	.	.	.	.	.	.

Table TC also reports a column showing an index of the cost of coding according to a simplified version of metrics adopted in [12], [9], [10]. This measure is obtained by evaluating the dimensions of locally defined class attributes, and it is useful for having an approximate measure of the effort needed for implementing, maintaining, reusing classes on the basis of the selected class hierarchy, and thus also for reengineering the code. In this case, the value of total cost is equal to 119. Please note that cost is equal to the dimensions

where no is-a relationship is defined. This value will be reduced by a suitable organisation of classes as demonstrated in the following.

In the example, the presence of COBOL clause `REDEFINES` in the definition of an elementary type means that the same memory area, allocated to a previously defined data structure, is also used by a new data structure. The new data structure could have a different meaning, but shares the same memory area. In the example `BIRTH-N` uses the same area as of `BIRTH`. This mechanism is frequently used by programmers for specifying that a variable employed like another variable with a mutual exclusive behaviour (e.g., in certain cases the social security number is used in the place of the number of driving licence). This relationship can be regarded as a specialisation/generalisation with attribute redefinition since the instances of the subclass can be substantially used as those of its superclass, but may have a different meaning or behaviour (methods) (e.g., checking of consistency of the code). For this reason, `BIRTH` class is considered the superclass of `BIRTH-N`. Moreover, since `PATIENT` class presents in its COBOL definition two different variables referring the same data area, a relationship of is-referred-by is established with class `BIRTH-N`. This means that in this specific case each instance of class `PATIENT` holds an instance of class `BIRTH` and a pointer to the same instance produced by class `BIRTH-N`, thus reproducing the early conditions.

3.4 Consolidation of Class Relationships

The consolidation phase of class hierarchy is performed by executing the following operations:

- simplification of the collection of elementary classes and instance variables;
- simplification of the collection of compound classes;
- identification of specialisation relationships among classes;
- identification of generalised classes.

This analysis consists in eliminating redundant instance variables in table TVI and unused classes in table TC. This process is drawn on the basis of the relationships among data structures reported in the other tables. As a first result, redundant instance variables in TVI are removed and the other tables will be updated accordingly; thus, TVI becomes:

Table of the Instance Variables, TVI			
TVI_ID	LABEL	Class_ID	Format_ID
1	`NAME`	{5}	1
2	`DAY`	{2}	2
3	`MONTH`	{2}	2
4	`YEAR`	{2}	2
5	`BIRTH-N`	{4}	3
6	`ID`	{3}	4
..	..	..	..

Please note that this table already reports the final values in the Class_ID column, and the is-part-of relationships between elementary classes and the other classes. In this phase, each entry of column Class_ID of table TVI is in practice a list of Class_IDs.

When classes with identical layout are detected, it is possible to eliminate directly the redundant classes, with the care of maintaining for the remaining class its synonyms. These are collected in the knowledge base of the eliminated classes. In this case the analysis can proceed automatically. On the other hand, this can lead to wrong assumptions, since structurally equal classes, which could be considered of the same type, can have very different behaviours. In these cases, the analysis of the functional aspects should help to take the final decision.

In our example, by forcing the equivalence of the identifiers `BIRTH` and `LAST-ORDER`, it is possible to consider those classes as unifiable, since they have the same dimension and equal instance variables. The redundant class is deleted from table TC.

When classes are equal "up to a certain point" it is possible to create a class hierarchy (is-a relationships) which aggregates in the superclass the common part and in the subclasses the variations and enrichments.

In the example both `PATIENT` and `CLIENT` classes share the definition of variable `NAME` as their part and, after the previous step, also that of `BIRTH/LAST-ORDER` class. Please note that class `BIRTH/LAST-ORDER` is used by modules 1 and 2. Therefore, it is possible to create a class (e.g., `Super-PATIENT-CLIENT`) to which the common parts are assigned and this will be considered as a superclass of the two previous classes. Class `Super-PATIENT-CLIENT` presents a reference to both modules in which their direct subclasses are present. This makes faster the phases in which the slices of code are assigned to classes. For the example, table TC has been updated as follows:

Table of the Classes, TC							
Class_ID	Class Name	Module_ID	is-part-of	is-a	is-referred-by	dim.	cost
1	`PATIENT`	$\{1\}$	-	$\{5\}$	-	46	0
2	`BIRTH/LAST-ORDER`	$\{1,2\}$	$\{5\}$	-	-	6	6
3	`CLIENT`	$\{2\}$	-	$\{5\}$	-	51	5
4	`BIRTH-N`	$\{1\}$	-	$\{2\}$	$\{1\}$	6	0
5	`Super-PATIENT-CLIENT`	$\{1,2\}$	-	-	-	46	46
..	..	-	-	-	-	..	..

In table TC, the final dimensions in bytes of class instances are reported. These values are substantially different from the theoretical dimensions of the classes which, by means of the is-a relationship, can be even defined without adding instance variables (such as `BIRTH-N` with respect to its superclass `BIRTH/LAST-ORDER`). In fact, with this new class hierarchy a strong reduction of cost has been obtained, thus producing a total cost equal to 57. Please note that this is less than half the value obtained at the beginning when no simplification on classes was performed.

This process of class organisation is typically performed also on code of modules. In fact, a correct identification of the class hierarchy helps to identify the duplications of slices of code and, thus, it avoids the generation of duplicated methods.

In Fig.4, the result of the final analysis, for the example discussed, is reported in the form of a class tree.

4 C_2O^2 and The Millennium Problem

The Millennium Problem (MP) is substantially a systematic implementation error due to the wrong acquisition of requirements in the early phases of many applications. Too much as been written on this

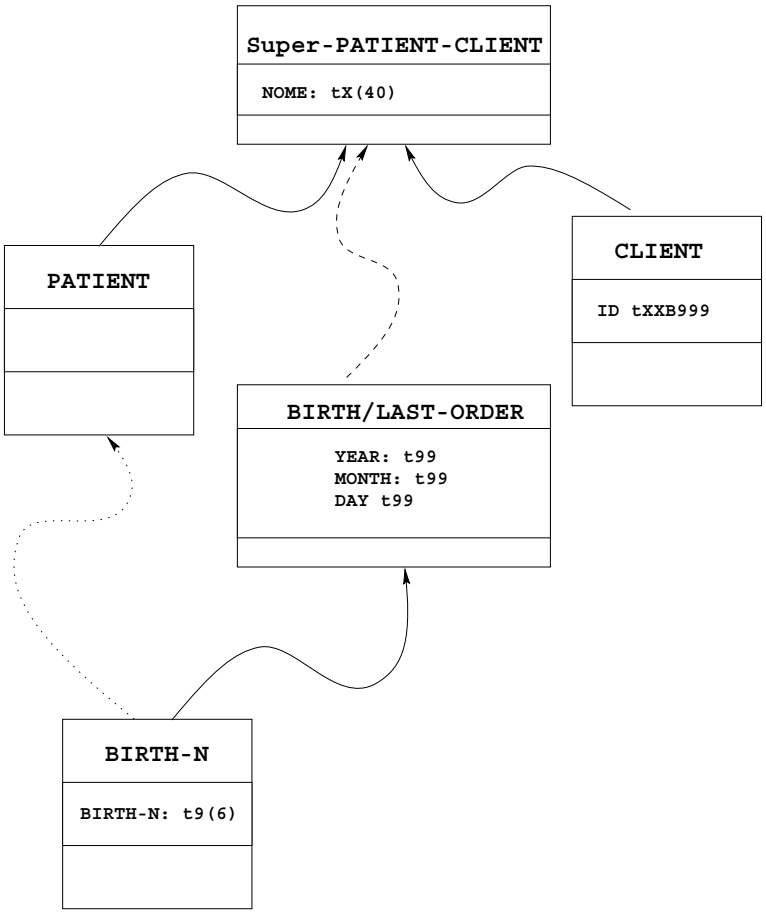

Figure 4: *Final class relationships: is-a as continuous lines, is-part-of as dashed lines, is-referred-by as dotted lines.*

problem to discuss it in few lines. Several opinions and discussions can be read by visiting www site `http://www.year2000.com/`.

From our point of view, the MP has to be considered as a relevant problem since there exists the effective possibility that the behaviour of an application is dependent on the date and, thus, its behaviour around the changing from 1999 to 2000 may lead to disasters or to unrecoverable errors. This is partially due to the frequent adoption of only two digits for storing dates, instead of 4 (e.g., in COBOL applications). In other languages, the date is frequently stored with an integer, thus the MP seems to be missing. On the other hand, the real problem is in the procedures that do not address the problem of changing from 1999 to 2000, for example for estimating the age, the delay of payment, the days of currency, the association of analyses in the hospital, etc. This can be really a problem in hospitals, in airports, and other organisations whose inefficiency could result even in injuries to people.

The MP could be considered less relevant for insurance companies, banks, service companies, etc. "Less relevant" in terms of human victims, but surely strongly relevant for its economical impact. These facts are leading most of the big companies to analyse their systems in order to identify and solve the MP in advance. This can be a mere target since most of the applications are capable of completing their work only by using other applications. These are in turn built by other companies, and so on. This means that the absolute confidence that the solutions adopted will make your system safe against the MP cannot be acquired, even if several test cases are made. On the other hand, this is true for every kind of error.

The above position is obviously contrasted by several technicians who are waiting for the error to begin to solve it. This can be true but disputable for small systems. For large systems, this cannot be accepted since in many cases the adoption of a different format for the date constrains to a reorganisation of databases, the modification of several hundreds of screens, print reports, and manuals. These operations cannot be performed as a simple, ordinary problem of maintenance [21].

On the contrary, an object-oriented analysis of similar applications usually leads to model the date as a unique class for the whole application domain. Therefore, a correct object-oriented reengineering of a COBOL application must imply the reduction of similar classes to a single one. This process is particularly exploited in our approach and tool.

This has been particularly evident in the first application of C_2O^2 which was the reanalysis of the software for managing university libraries on a national basis. It is called SBN (Servizio Bibliotecario Nazionale, National Library Service) and consists of 2184 modules for a total of more than ten megabytes of COBOL code, with much as 378,767 lines of obscure and old-fashioned COBOL crafted by more than 30 developers (about 1 Million of keywords and labels), without a constant evolution plan over eight years of use, with only 27193 comments.

In SBN software, several thousands of instances of the MP have been found; in fact all the modules have only two-digit-year in the date definitions. Other systematic errors can be found in the procedural code, especially when the date is read from the system clock.

The process of COBOL code analysis has identified a huge amount of data structures containing dates. For example, in 1376 main modules, the early version of TVI resulted in 36189 entries, and the table of classes, TC, in 10314 entries, with 484 different formats. There were more than 1500 instances of dates in 9 different formats.

The typical conceptual process which C_2O^2 follows for enlightening the problem – first the format and then the label unification – has allowed to classify 9 data structures as instances of a same class, `DATE`

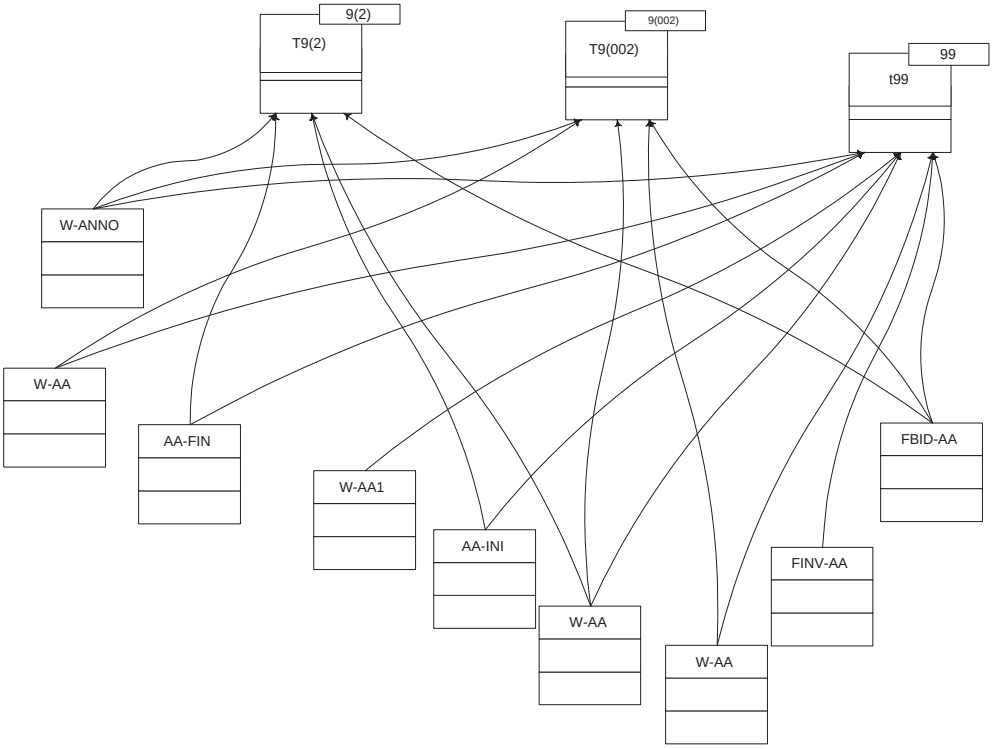

Figure 5: *Early classes for formats and instance variables modeling years, with their relationships.*

parameterised on the physical storage format. Please note the similar names of the variables. In Fig.5, the relationships among the early classes modelling the formats and those which model the variables representing years into the SBN application are reported. According to steps 2 and 3, the unification of formats and that of labels reduce the collection of classes reported in Fig.5 in a simple couple of classes as depicted in Fig.6.

In this case, it is really simple to understand that a reduction of classes means also a reduction of the effort in making modifications for solving the MP.

By using C_2O^2 the passage from the early implementation to an object-oriented description of the domain problem was performed by means of a reasoning-based iterative process.

5 C_2O^2 : A Tool for Reengineering COBOL Applications

C_2O^2 (COBOL to Object-Oriented tool) is an instrument for analysing COBOL applications by means of the OOP. The analysis performed by C_2O^2 tool is focussed on modelling the problem domain according to the OOP.

C_2O^2 is based on a Lex/Yacc engine which is capable of processing all COBOL syntax and semantics (the current prototype is able to recognise a particular version of COBOL close to the standard, but provisions have been included to customise it to other versions). This tool is supported by a set of predefined classes for modelling the main entities which are typically present in COBOL applications. Fig.7 shows the relationships among class `Database` of the tool class library, class `SBN` obtained for collecting all database features collected for the SBN application, and the main classes which model the record to be stored into the database. Class `Database` provides all methods for accessing to database.

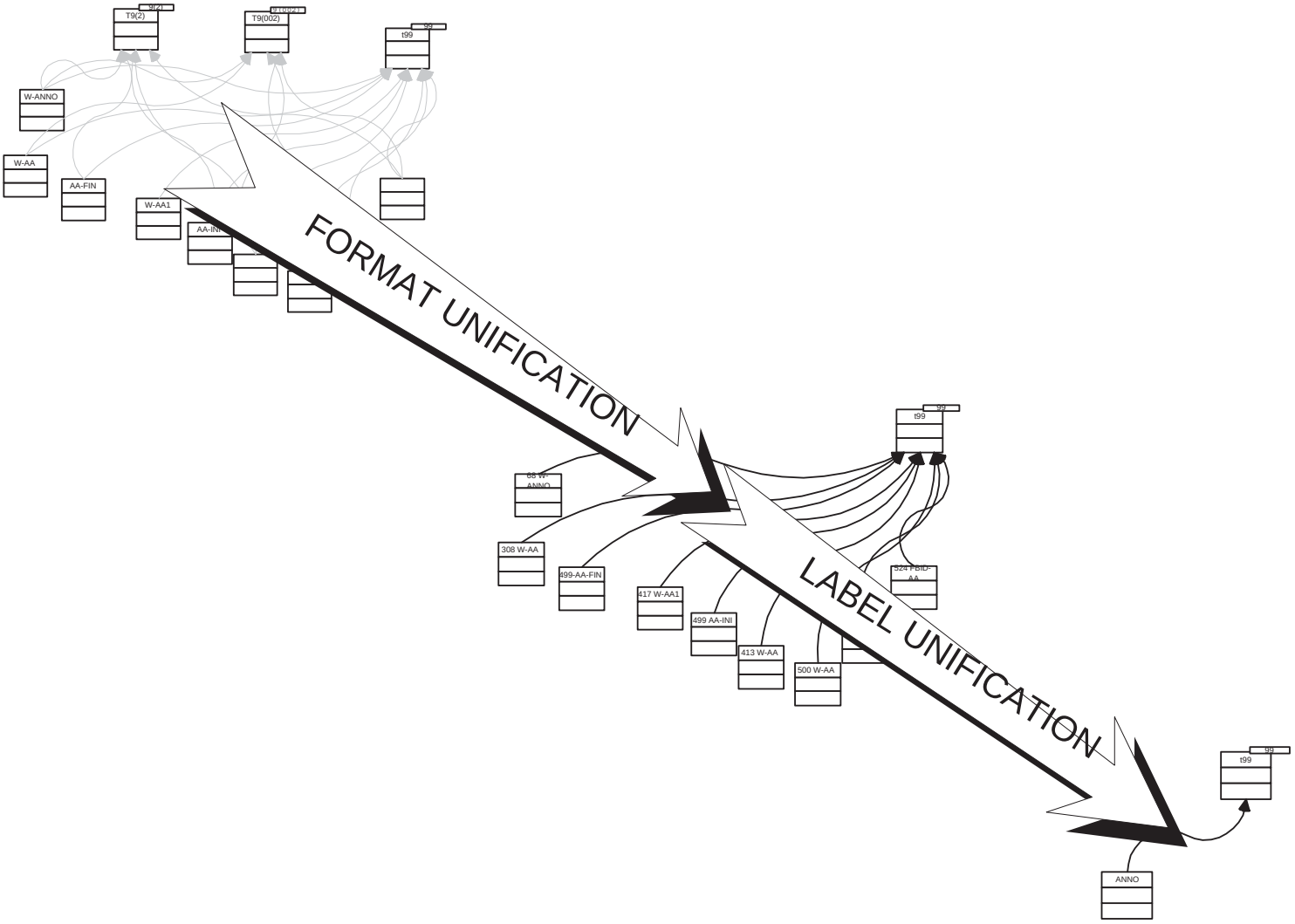

Figure 6: C_2O^2 *process for enlightening Millennium Problems into the SBN.*

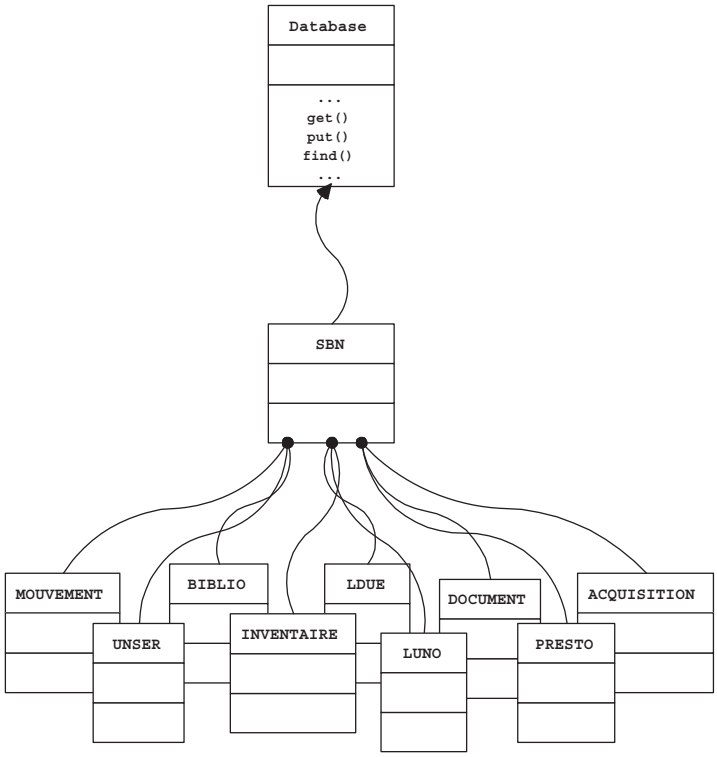

Figure 7: *Relationships among class* `Database` *of the tool and the related application classes.*

Fig.8 depicts the relationships among class `File` of C_2O^2 class library and `FINV` and `FBID` classes (partially reported in the early example), and class `DATE`. Class `File` contains all methods and attributes to perform a general access to files.

Due to the high number of similar and identical data structures which are usually present in COBOL applications, the process of data structure unification needs to be practically supervised by the user. The process of application domain analysis is performed by means of the definition of a knowledge base containing synonyms or by direct assistance from the user. When similar data structures identified by C_2O^2 are missing in the knowledge base, the final decision about their unification is left to the user. Thus, an iterative process of continuous improvement is established in order to minimise the complexity of the reengineered application. This process is also assisted by some simple metrics based on class size.

The objective of the software prototype we have developed was to show the practical feasibility of the reengineering trajectory sketched in the previous sections. The prototype is capable of performing the recognition, abstraction and transformation of concepts, supporting different reengineering methods, supporting multiple models, providing visual and textual multiple but consistent representations, and of providing an explicit support to software evolution techniques.

The features described are made accessible to the user by means of two main activities of C_2O^2: (i) the automatic abstraction, using the algorithm shown in the previous section and (ii) the interactive modelling, which allows the updating of the models of the source code under examination, following an evolutionary strategy, according to the user's needs. The models are collections of meaningful activities, obtained by the analysis of the source code. Once it has been created, a model can be refined by the user by means of commercially available specific evolutionary design tools.

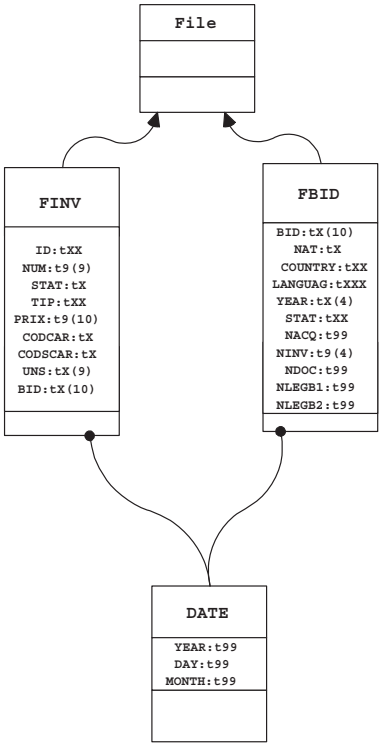

Figure 8: *Relationships among class* `File` *of the tool and the related application classes.*

6 Discussions and Conclusions

In this paper, a method/tool (C_2O^2, COBOL to Object-Oriented) for analysing COBOL applications has been presented. This tool is capable of identifying classes and their relationships by means of a process of understanding and refining in which COBOL data structures are analysed, aggregated, and simplified semi-automatically. It can support different reengineering methods, and has been used on a large application such as the software for managing all libraries of the University of Florence (where many instances of the millennium problem were found) with very interesting results. Work is in progress for implementing the mechanism for the automatic association of procedural parts to classes.

Similar approaches and tools for migrating COBOL applications towards object-oriented models have been proposed in the literature. One of the first methods offered for migrating COBOL applications towards object-oriented ones has been REDO [18]. This was mainly oriented to derive from a COBOL application its specification in Z++ [22]. REDO is capable of converting BATCH COBOL programs into an intermediate language and from this to Z++. Keywords REDEFINES of COBOL are converted into invariant assertions, while every COBOL record is converted into a class having as attributes its data components. Program slices which use class attributes are assigned to classes as methods. In this approach, no elimination of redundant definitions of formats as well as of duplicated classes and methods (e.g., by defining generalised classes) is performed.

The MOORE approach/tool was presented in [20] for converting COBOL applications into Object COBOL. It works only on pure COBOL85 programs and is capable of converting COBOL data structures in classes by means of a user-guided process in which the MOORE tool proposes solutions to be selected by the user. In this process only little attention is given to a full analysis of the application and

to establishing optimal relationships among classes in order to minimise the effort of code reuse. In fact, the typical COBOL duplication of code and data definitions is also produced into the new version of the application.

The REDOC project [19], [6] is based on the assumption that classes can be directly derived from modules defining SUBprograms, while ONLINE and BATCH modules contain references to several classes and are far from being regarded as single classes. These assumptions could be true for some modules, but in general the SUBprograms, as well as the methods in object-oriented applications, work on local attributes and temporary instances of other classes. For this reason, the early basic assumption chosen can lead to a less satisfactory identification of classes and, thus, of the object-oriented analysis of the application domain.

CORECT (`http://www.acm.co.uk`) is a commercial tool for migrating COBOL applications towards Java and other languages. Even with CORECT tool, the main problem we have experienced with the version tested is the lack of mechanisms for automatically reducing equivalent formats (elementary types of the applications). This leads to an excessive number of classes. CORECT tool is also less satisfactory in defining the relationships of specialisation; thus, the large duplication of data and code is also maintained in the object-oriented version of the application. If the user pays attention to prepare the conversion eliminating the duplications of formats, then a strong improvement of results is obtained.

As a conclusion, differently from the methods reviewed, the method proposed is, in our opinion, superior in:

- making an object-oriented analysis of the problem domain of the COBOL application;
- reducing the structural complexity of the application by reducing the number of elementary types (reduction of formats);
- increasing the reusability and the maintainability of system classes by organising classes according to the OOP and identifying generalisations.

These problems were stressed in [19] and [23] as the main drawbacks experienced in the migration of COBOL applications towards object-oriented. With our approach for reducing multiple formats and identifying the class hierarchy, we have partially solved these problems. The production of a worst object-oriented analysis of the COBOL application can lead to high costs of reuse and maintenance; an object-oriented system with a bad class organisation can be even worst than an old procedural application.

Acknowledgements

The authors would like to thank Dr. Ing. E. Somma for the early experiments of the project and tool discussed.

References

[1] G. Bucci, M. Campanai, and P. Nesi, "Tools for specifying real-time systems," *Journal of Real-Time Systems*, vol. 8, pp. 117–172, March 1995.

[2] T. J. Briggerstaff and A. J. Perlis, *Software Reusability: Volume I, Concepts and Models*. New York: Addison Wesley, ACM Press, 1989.

[3] J. Faget and J.-M. Morel, "The REBOOT approach to the concept of a reusable component," in *Proc. of 5th Annual Workshop on Software Reuse, WISR'92*, (Palo Alto, CA, USA), 26-29 Sept. 1992.

[4] O. Signore and M. Loffredo, "Some issues of object-oriented re-engineering," in *Prof. of ERCIM Workshop on Methods and Tools for Software Reuse*, (Heraklion, Crete, Greece), 1992.

[5] E. J. Chikofsky and J. H. C. II, "Reverse engineering and design recovery: A taxonomy," *IEEE Software*, vol. 7, pp. 13–17, Jan. 1990.

[6] H. M. Sneed and E. Nyary, "Extracting object-oriented specification from procedurally oriented programs," in *Proc. of the 2nd Working Conference on Reverse Engineering*, (Toronto, Ontario, Canada), pp. 217–226, IEEE Press, July 14-16 1995.

[7] G. Booch, *Object-Oriented Design with Applications*. California, USA: The Benjamin/Cummings Publishing Company, 1994.

[8] P. Nesi, *Objective Software Quality, Proc. of Objective Quality 1995, 2nd Symposium on Software Quality Techniques and Acquisition Criteria*. Berlin: Lecture Notes in Computer Science, N.926, Springer Verlag, 1995.

[9] P. Nesi and M. Campanai, "Metric framework for object-oriented real-time systems specification languages," *The Journal of Systems and Software*, vol. 34, pp. 43–65, 1996.

[10] P. Nesi and T. Querci, "Effort estimation and prediction of object-oriented systems," *The Journal of Systems and Software*, vol. in press, 1998.

[11] J. Daly, J. Miller, A. Brooks, M. Roper, and M. Wood, "Issues on the object-oriented paradigm: A questionnaire," tech. rep., Dept. of Computer Science, Univ. of Strahclyde, UK, RR-95-183, June 1995.

[12] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics, A Practical Guide*. New Jersey: PTR Prentice Hall, 1994.

[13] E. C. Arranga and F. P. Coyle, *Object-Oriented COBOL*. SIGS BOOKS & MULTIMEDIA, 1996.

[14] C. L. Ong and W. T. Tsai, "Class and object extraction from imperative code," *Journal of Object Oriented Programming, JOOP*, 1993.

[15] I. Jacobson and F. Lindström, "Re-engineering of old systems to an object-oriented architecture," in *Proc. of OOPSLA 1991, 6th Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM SIGPLAN NOTICES VOL. 26, N.11*, (Phoenix, USA), October 1991.

[16] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. New Jersey: Prentice Hall International, Englewood Cliffs, 1991.

[17] S. Pidaparthi and G. Cysewski, "Case study in migration to object-oriented structure using design transformation methods," in *Proc. of the 1st Euromicro Conference on Software Maintenance and Reengineering*, (Berlin, Germany), pp. 128–135, IEEE Press, 17-19 March 1997.

[18] P. T. Breuer and K. Lano, "Creating specifications from code: Reverse-engineering techniques," *Software Maintenance: Research and Practice*, vol. 3, 1991.

[19] H. M. Sneed, "Migration of procedurally oriented COBOL programs in an object-oriented architecture," in *Proc. of the IEEE Conference on Software Maintenance*, (Orlando, Florida), pp. 105–116, IEEE Press, Nov. 9-12 1992.

[20] H. Fergen, P. Reichelt, and K. P. Schmidt, "Bringing objects into COBOL: MOORE - a tool for migrating from COBOL85 to object-oriented COBOL," in *Proc. of the International Conference on Technology of Object-Oriented languages and Systems, TOOLS USA 94*, pp. 435–448, 1994.

[21] B. Muller and R. Gimnich, "Planning year 2000 transformations using standards tools: An experience report," in *Proc. of the 1st Euromicro Conference on Software Maintenance and Reengineering*, (Berlin, Germany), pp. 94–100, IEEE Press, 17-19 March 1997.

[22] K. Lano and H. Haughton, *Object-Oriented Specification Case Studies*. New York, London: Prentice Hall, 1994.

[23] A. Fantechi, P. Nesi, and E. Somma, "Object-oriented analysis of COBOL," in *Proc. of the 1st Euromicro Conference on Software Maintenance and Reengineering*, (Berlin, Germany), pp. 157–164, IEEE Press, 17-19 March 1997.

Contents