Data model reverse engineering in migrating a legacy system to Java

Mariano Ceccato$^{(1)}$, Thomas Roy Dean$^{(2)}$, Paolo Tonella$^{(1)}$, Davide Marchignoli$^{(3)}$

(1) FBK-IRST, Trento, Italy

(2) Queen's University, Kingston, Canada

(3) IBT, Trento, Italy

ceccato@fbk.eu, tom.dean@queensu.ca, tonella@fbk.eu, davide.marchignoli@ibttn.it

Abstract

Central to any legacy migration project is the translation of the data model. Decisions made here will have strong implications to the rest of the translation. Some legacy languages lack a structured data model, relying instead on explicit programmer control of the overlay of variables. In this paper we present our experience inferring a structured data model in such a language as part of a migration of eight million lines of code to Java. We discuss the common idioms of coding that were observed and give an overview of our solution to this problem.

1 Introduction

An integral part of maintenance is the migration of existing systems to use new technology. There are many reasons for the use of the new technology, one of which is the obsolescence of part or all of the current implementation platform such as the implementation language. While some languages, such as COBOL, remain widely used within their domains, others of more limited deployment may become a liability to future maintenance. This paper describes part of a project at FBK-IRST to migrate a terminal based legacy banking system written in a proprietary language to a Java based application server (a decision made by the customer).

The language used by the legacy system is BAL, an acronym for Business Application Language. BAL is a BASIC like language that contains unstructured data elements (described in Section 2) as well as unstructured control statements (e.g., `GOTO`). Programs are composed of multiple segments and may also contain user defined functions. Calls between programs are supported, and a preprocessor provides the programmer with the ability to isolate common code in files that may be included in more than one program.

As with any large scale migration project, there are several goals that are, in a sense, mutually contradictory. The first of these is that we wish to preserve the familiarity that the developers have with the existing code base. That is, the relation between the original BAL source and results of the translation should be visible, and a developer responsible for maintaining the original BAL should be able to identify the locations and implementations of concepts in the resulting Java code with little difficulty. The second goal is that the code be high quality, idiomatic Java. That is, a naive translation that implements the semantics of the BAL programs in Java syntax would be difficult to understand and to maintain.

Central to any translation effort is the translation of the data model. While applications coded in legacy languages such as BAL are based on a functional decomposition approach, the object oriented design method focuses on the data model. In this paper we describe an approach using program transformation to reverse engineer a structured data model from the unstructured model provided by BAL.

The starting BAL data model allows programmers to overlay variables in memory with (almost) arbitrary layouts. Memory can be regarded as a byte array and the position of each variable in this byte array can be specified relative to other, previously declared variables. Such a data model is quite close to that of Assembly, where memory addresses can be directly used to specify the location of variables. All modern programming languages, including Java, but also Cobol and similar languages, restrict the admitted overlays of variables in memory, so as to enforce some notion of containment. An object in Java may contain attributes, which in turn may reference objects, but no two objects are ever aliases of each other in Java or have different, partially overlapping displacements in memory. A record in Cobol contains fields, which can be in turn records, but field containment cannot be broken by arbitrary, "unstructured" variable declarations. Hence, in a migration effort to a modern language, the first problem being faced with BAL is reverse engineering of a structured data model starting from the current, unstructured one. The next steps, out of the scope of the present work, would be aggregating structured data and operations into a "true" OO model and inferring fine grained objects, such as dates, by aggregating individual variables based on use.

The rest of the paper is organized as follows: In section 2, we describe the basic cases that may occur in BAL code and how we map them to Java. Unfortunately, the permissive data model of this language allows programmers to deviate from such basic cases. Exceptions are described in Section 3, where we explain how these cases are currently managed. Section 4 provides empirical data on the occurrence of the various cases in the particular system we are migrating. In Section 5, we describe some of the previous work in the area and conclude in Section 6.

2 Base strategy rules: exact size match

In this section we give a short introduction to the data model provided by the BAL language and provide some examples of the basic ways in which the conventional notion of records and fields are expressed in the BAL language.

```
DCL a#       // Byte Variable
DCL b%       // Short Variable
DCL c&=5     // BCD Variable, 5 bytes long
DCL d$=100   // String Variable, 100 bytes long
DCL e$       // String Variable, 16 bytes long
```

Figure 1. Primitive types

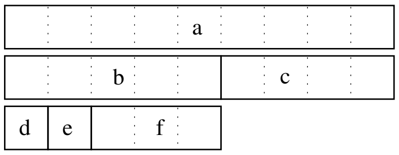
While BAL contains some structured control flow statements such as `IF... ENDIF` and `WHILE... WEND`, the data model is very unstructured and similar to that found in structured assembly languages (e.g., that of IBM mainframes). The data model is byte oriented and the language only provides four basic data types: byte, short, binary coded decimal (BCD) and string. The first two are the same as those available in most languages, representing a single byte and two contiguous bytes respectively. Variables of the BCD and string data types can be of different lengths, and the developer must specify the length (in bytes) if she wants something different than the default length. Unlike languages such as C, there is no dynamic allocation, and the length of all variables is known at compile time.

Figure 1 shows a simple example of variable declarations. The variables `a` and `b` are byte and short variables (indicated by the type specifier '#' and '%'). The variable `c` is a BCD variable(type specifier '&', optional) that takes five bytes of storage. BAL stores the BCD value in its own, proprietary format. The variable `d` is a string variable one hundred bytes long. In the absence of an explicit length (i.e. "= <*expression*>"), default lengths of eight bytes for BCD variables and sixteen bytes for string variables are used. Thus the variable `e` is a string variable that is sixteen bytes long. Arrays of each of the types are also supported by the language, with at most two indexes (i.e., either vectors or matrices).

Even mapping the atomic BAL types to Java types is not straightforward. `Byte` and `short` have a natural counterpart in Java, even though using `byte` and `short` in Java introduces downcasts, since intermediate computations may get automatically promoted to `int`. BAL strings are different from Java strings in a few respects: they are represented as byte (8 bit) sequences, not as UNICODE character (16 bit) sequences, they are mutable and they have fixed length. The mapping with the closest semantics would be the Java byte array. However, translation of BAL strings into byte arrays would result in low quality, poorly maintainable Java code, in that it would deviate from the common Java programming practice and it would need ad hoc support for manipulation of the translated strings. Instead of resorting to an ad hoc data type based on a byte array representation, we decided to use the Java type `String` anyway, by providing proper translations and helper functions, when needed. Modifications of BAL strings are translated into reassignments and proper helper functions are provided for string truncation or padding up to the length declared in BAL. Such helper functions take advantage of annotations that record the original BAL string size. BCD numbers can be mapped to the `BigDecimal` type in Java, but again some care must be taken. As with BAL strings, the size of the BCD must be recorded in annotations. `BigDecimals` are also immutable, hence reassignment is needed whenever a BAL BCD is modified. Rounding rules should replicate exactly the same semantics as in BAL, so the appropriate mathematical context (`MathContext` object) should be chosen for all generated `BigDecimals`, as well as for the intermediate arithmetics.

2.1 Simple overlay

In BAL, variables are laid out sequentially in memory, with global variables in the global space and local variables on the data stack. Grouping of variables into records is done by explicitly overlaying variables by giving them overlapping positions in memory. This is accomplished with the `FIELD=M,`*VAR* statement, as shown in Figure 2. The code starts by declaring a string variable `a` of length nine. The `FIELD` statement resets the current variable position (i.e. the position of the next declared variable) in memory to the beginning of the variable `a`, and as a result, the string variable `b` has the same starting position as the `a`, but a shorter length. The variable `c` which follows `b` is assigned to the next location in memory after `b`, which is also within the boundaries of the variable `a`. In fact, both variables (total length of nine bytes) are contained within variable `a`. Thus an assignment to the variable `a` will also change both `b` and `c`, while an assignment to `b` will only change the first five bytes of the variable `a`.



```
#ifdef A
DCL a$ = 9
FIELD = M, a
  DCL b$ = 5
  DCL c$ = 4
  FIELD = M, b
    DCL d#
    DCL e#
    DCL f$ = 3
#endif
```

```
a DS CA9
b EQU a
  DS CA5
c DS CA4
d EQU b
  DS B1
e DS B1
f DS CA3
```

Figure 2. Simple containment with exact size match

The second FIELD statement resets the current variable position to the beginning of b (which is also the beginning of a), and the three variables, d, e and f are all allocated from that position. Figure 2 (top) shows the position of variables in memory diagrammatically. Using the FIELD=M statement without a variable name resets the current variable position to the first position free in memory. In our example, if appended at the end of the declarations, such a statement would move the next data position available in memory immediately past the end of variable a, since all of the other variables are located within the space allocated to a. The right hand listing in Figure 2 shows an equivalent data structure in mainframe assembly language (DS = allocate data storage, CA=ASCII string, B1=binary byte). The EQU directive is the equivalent of the FIELD=M,*VAR* statement.

As can be seen in the figure, C style preprocessing statements are available to the developer. Data structures are kept in separate files (some of which automatically generated from ISAM[1] tables) that are included using the #include directive. Macro definitions are used to select (via #ifdef) which data structures to instantiate (e.g., #ifdef A, in Figure 2).

There are several consequences to the approach taken by the BAL language. The first consequence is that records do not introduce any additional lexical scope: the name space is flat and there is no equivalent of the dot notation (e.g. a.b), common in languages such as C and Java. The second consequence is that it is the developers' responsibility to ensure that the sizes of the variables are correct. For example, in Figure 2 above, the variable a is intended to be a reference to the entire record. If the size of c is changed to five, then the size of a should also be changed. The last consequence is that there are many ways of expressing the exact same layout of variables within memory. The last two consequences make the recovery of a structured record from a sequence of BAL declarations difficult.

[1](ISAM, Indexed Sequential Access Method – a data file format common in legacy systems)

```
public class A {
    Aa a = new Aa();
    class Aa {
        Ab b = new Ab();
        class Ab {
            byte d;
            byte e;
            @Field(size=3)
            String f;
        }
        @Field(size=4)
        String c;
    }
}
```

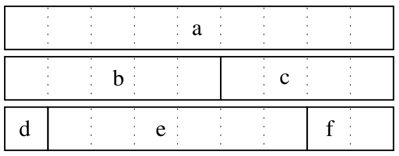
Figure 3. Simple containment

For the record data type, which is obtained in BAL through the FIELD=M construct, the mapping to Java is straightforward in case of simple containment with exact size match (as depicted in Figure 2). Figure 3 shows the Java code produced for the example in Figure 2. Nested FIELD=M instructions are mapped to inner classes in Java. BAL strings used as record containers become Java objects, the type of which is the Java class corresponding to the FIELD=M defined upon them. For example, the BAL strings a and b in Figure 2 are turned into the two objects a and b, declared as class attributes within class A and Aa, and initialized with instances of class Aa and Ab respectively.

The translation shown in Figure 3 makes the assumption that records are either accessed through their fields (the leaves of the containment tree) or, as a whole, through the container itself, used as a reference to the record. For example, if field b is read or written as a BAL string in the BAL code, the generated Java object must resort to serialization methods (e.g., readFrom and writeTo) to properly assign values to its attributes.

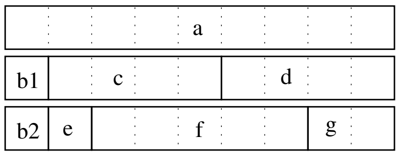
2.2 Multiple overlay

As with many legacy applications, the developers sometimes use alternate views of the same memory. The root cause of this descends from the persistence layer, in our case ISAM tables, where multiple record types are often hosted inside the same table for performance optimization reasons or just because it is permitted by the language. In the source code, this turns out to be similar to the union construct, provided by languages such as C and C++. Figure 4 shows an example. Variable a, with length 9, has been redefined twice. Once by two strings b and c. The other by three variables, d, e and f. The variable d is a byte, while the variable `f` is a short. The variable `e` is a three element array of strings, where each element has a length of two. An assignment to the variable `b` will change the values of the variable `d` and the first two elements of the array `e`.



```
DCL a$ = 9
FIELD = M, a
   DCL b$ = 5
   DCL c$ = 4
FIELD = M, a
   DCL d#
   DCL e$ = 2(3)
   DCL f%
```

Figure 4. Union



```
#ifdef A1
DCL a$ = 9
FIELD = M, a
   DCL b1$ = 1
   DCL c$ = 4
   DCL d$ = 4
#endif
```

```
#ifdef A2
#ifdef A2_SKIP
DCL a$ = 9
#endif
FIELD = M, a
   DCL b2$ = 1
   DCL e#
   DCL f$ = 5
   DCL g%
#endif
```

Figure 5. Mutually exclusive overlays

A special case of the union data structure is characterized by mutually exclusive overlays. In this case, one or more bytes of the structure form a *discriminator*, which identifies which overlay is intended to be used. One situation in which this variant is used is when reading or writing a table in which multiple record types are stored. Figure 5 shows an example of such a data structure. The two variants of the storage are the variables `b1`, `c` and `d` on one hand, and the variables `b2`, `e`, `f` and `g` on the other. The two data structures can be instantiated individually (either by defining only the macro `A1`, or by defining only the macros `A2`, `A2_SKIP`). In such cases the data structure has only one view active (i.e., it is not a union). Union instantiation is achieved by defining both `A1` and `A2`, while leaving `A2_SKIP` undefined, so that the second group of declarations (on the right in Figure 5) overlays with the first one.

When a union is instantiated, the string variables `b1` and `b2` align (i.e., reference the same memory position) and comprise the discriminator of this record. One value, say the value `"T"`, will indicate that the first variant is to be used, while another, say the value `"D"`, will indicate that the other variant is valid. The BAL language does not enforce mutually exclusive access to the record variants. It is up to the developer to code the related logic appropriately, by making sure that every access according to one of the views defined for the given union is guarded by some instruction ensuring the discriminator holds the value corresponding to the view being used.

Unions have no obvious counterpart in Java. The idea behind unions is that an object is made accessible through multiple views. In Java, one way to express such multiple accessibility can be achieved by making the object implement multiple interfaces, each of which associated with one of the multiple views. In order to avoid replication of data, the union object may implement the copy-on-read/write protocol, which allows lazy creation and update of the alternative views available from the object.

Figure 6 shows the translation of the union in Figure 4. Class `UnionAa` implements the two interfaces associated with the two alternative views defined in Figure 4 for variable `a`. The first view exposes the fields `b` and `c`, hence the related interface (`Aa1Int`) has getter and setter methods for the corresponding class attributes `b` and `c`. Similarly, the second interface will expose getters and setters for `d`, `e` and `f` (not shown in Figure 6 for space reasons). Since `UnionAa` implements both interfaces, it must expose getters and setters for all fields in all alternative views (i.e., `b, c, d, e, f`).

Lazy creation and update of the alternative views in a union is achieved by initializing the union fields for the variants to *null*. In Figure 6, inside class `UnionAa` both attributes `a1` and `a2` are initialized to *null*. When a setter or getter is invoked on the union object, a `switchVariant` operation is invoked if the current active variant of the union is different from the requested one. Then, the set or get operation can be delegated to the proper object (`a1` or `a2` in our example). The `switchVariant` operation has responsibility for creating the requested variant, if the related attribute has *null* value, and for copying the field values from any other non-*null* variant, in case it exists. The `switchVariant` operation ensures that at each point in time only one union variant has non-*null* value, so it must also take care of assigning *null* to the copied non-*null* variant, when it is there.

With reference to Figure 6, if `setB` is called and both `a1` and `a2` are null, the `switchVariant` method will create an `Aa1` object and assign it to `a1`. If `setB` is called and `a2` is non-*null*, `switchVariant` will copy all fields of `a2` into fields of `a1`. Since fields may be not aligned and of different type, field copy from one variant to another one resorts to the serialization operations `readFrom(Reader)` and `writeTo(Writer)` (not shown in Figure 6 for space reasons), to be used whenever a switch from one union variant to another one occurs.

```
public class A {
    UnionAa a = new UnionAa();
    class UnionAa implements Aa1Int , Aa2Int {
        Aa1 a1 = null; // lazy creation
        Aa2 a2 = null; // lazy creation
        String getB() {... return a1.getB();}
        void setB(String b) {
            if (a1 == null) switchVariant(...);
            a1.setB(b);
        }
        ...
    }
    class Aa1 implements Aa1Int {
        @Field(size=5)
        String b;
        String getB(){...}
        void setB(String b){...}
        @Field(size=4)
        String c;
        String getC(){...}
        void setC(String c){...}
    }
    class Aa2 implements Aa2Int {...}
}
interface Aa1Int {
    String getB(); void setB(String b);
    String getC(); void setC(String c);
}
interface Aa2Int {...}
```

Figure 6. Union

When the different views of a data structure are mutually exclusive, we can take advantage of inheritance and we can instantiate the appropriate subclass, instead of resorting to unions. In Figure 5, the value of `b1` (or equivalently `b2`) determines the record type. Whenever `b1=="T"`, the first view is accessed, while `b1=="D"` selects the second view. In Java, the discriminator is named `b` and is moved to the common superclass `A` (see Figure 7). The value of the discriminator in the code determines which subclass of `A` to instantiate or which downcast to use on an object of type `A`. For example, if a BAL code portion instantiates the data structure in Figure 5 assigning the value `"T"` to `b1`, we know the Java translation must instantiate class `A1`. If an object has type `A` (e.g., because it is returned by a BAL function), but all its uses are guarded by `b1=="D"`, we can downcast it to `A2` and use the specific methods of `A2` in the translated code.

3 Exceptions to the basic rules

In this section, we examine variable declarations in BAL that deviate from the basic cases described in the previous section. For each case, we describe how we manage to reverse engineer a structured representation of the data. Since cases have been discovered heuristically and do not cover the entire set of possibilities offered by BAL, there

```
public class A {
    @Field(size=1)
    @Discriminator()
    String b; // was: b1, b2
}
public class A1 extends A {
    @Field(size=4)
    String c;
    @Field(size=4)
    String d;
}
```

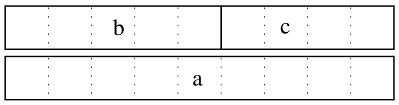
```
public class A2
        extends A {
    byte e;
    @Field(size=5)
    String f;
    short g;
}
```

Figure 7. Mutually exclusive overlays

is a chance that none of the cases presented in this section applies, with the consequence that our reverse engineering technique fails and manual intervention is required. Manual intervention is also required when a known problem is recognized automatically, but we have no automated solution for it (e.g., missing container described below). The amount and cost of such manual interventions are assessed empirically in the next section.

3.1 Inversion

Figure 8 represents a common alternative way of expressing the same top level structure as shown in Figure 2 (w.r.t. variables `a`, `b`, and `c` only). In this example, the developer has first specified the sequence of fields in the structure before overlaying the fields with a single larger variable which is used to reference the fields as a whole. The overall layout in memory, however, remains the same.



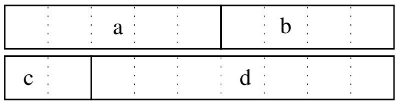
```
DCL  b& = 5
DCL  c$ = 3
FIELD = M, b
  DCL a$ = 9
```

Figure 8. Inversion

The case of a `FIELD=M` with inversion (Figure 8) is mapped to Java similarly to simple containment (see Figure 3), once the container has been recognized. The heuristics to recognize an inversion is the following: a `FIELD=M` instruction refers to a variable (e.g., `b`) smaller than the first following declaration (`a`). Then, the exact size match condition is verified assuming the redefiner (`a`) is the container and the redefinees (`b`, `c`) are the record fields. If the sum of the sizes of the redefinees is equal to the size of the candidate container (redefiner), an inversion is detected and mapped to the Java class described previously (see Figure 3).

3.2 Missing container

Existence of a container for the entire record is neither enforced nor necessary in BAL. In fact, the first field of the record can be used as a reference to the beginning of the record and a `FIELD=M` instruction, followed by a list of declarations that exceeds the field size, can be used to access the the full record. An example of this programming style is shown in Figure 9. This data structure is a union for which no container variable is defined. The two views available in this union (either a record with fields `a, b`, or a record with `c, d`), are accessed through the first record field (either `a` or `c`), even though its size is less than the entire record size. Access to the next fields (e.g., `b`) is easily achieved via `FIELD=M,a` followed by proper declarations (e.g., `DCL aa$=5, DCL b$=4`).



```
#ifdef A
DCL a$ = 5
DCL b$ = 4
FIELD = M, a
  DCL c$ = 2
  DCL d$ = 7
#endif
```

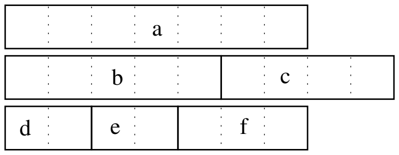
Figure 9. Missing container

Currently, we have no heuristics to manage this exception. The exact size match condition is clearly violated and no inversion can be detected. As a consequence, our tool reports a size mismatch error to be fixed manually. The manual fix consists of adding a surrounding container to the declarations shown in Figure 9, e.g., `DCL aa$=9` before the declaration of `a`, which is turned into a redefinition of `aa`. Once the missing container has been added, mapping to Java of this example can be achieved along the lines described in the previous section for unions (see Figure 6).

3.3 Wrong container size

Sometimes, the container for the whole record may exist but it may have the wrong size. In fact, it is the programmer's responsibility to indicate the size of all variables, including those that act merely as containers. If, during software maintenance, any field size changes, the change must be propagated to all container variables for the changed field. Such a propagation is manual in the source code, while it is tool-supported for the code generated automatically from ISAM tables. In both cases, the programmer is in charge of performing the size update. In most cases, while the compiler does not complain, if enough memory is allocated for the data structure, no run-time error ever shows up. So, from the point of view of BAL programming, it is acceptable. However, recognizing a single data structure with a container may become difficult in such a situation.

Figure 10 shows an example where the declared container size is 7 instead of 9. Apparently, half of variable `c` is declared inside the data structure, while half of it is part of the next free memory positions. This is the typical hint of a wrong container size. However, it may be hard to determine how many declarations following `b` should be attributed to the record `a`. Consistent declaration of fields for a total size not exceeding 9 in the alternative views of this union indicates that the correct container size is probably 9 in this example.



```
#ifdef A
DCL a$ = 7
FIELD = M, a
  DCL b$ = 5
  DCL c$ = 4
FIELD = M, a
  DCL d$ = 2
  DCL e$ = 2
  DCL f$ = 3
#endif
```

Figure 10. Wrong container size

The heuristics to recognize all cases of wrong container size are detailed in the next subsection. In the example in Figure 10, once we are able to recognize that the container size should be incremented, we can apply the same translation used for the basic case shown in Figure 4, resulting in a Java class similar to the one in Figure 6.

3.4 Other mismatching cases

Before producing the Java object model, we encode the reverse engineered data model in the form of square brackets surrounding all declarations in the scope of a given `FIELD=M`. Moreover, multiple `FIELD=M` referring to the same declarations are enclosed within square-angular brackets. Figure 11 shows the output of this reverse engineering step for the examples of BAL code in Figures 2 and 10.

The containment relationship represented by means of square brackets is computed by comparing the size of the original variable (i.e., `a`) with the sum of the sizes of the variables in its overlay (i.e., variables after `FIELD=M,a`).

The problem with producing the correct declaration bracketing is that the size of the different memory overlays often does not match. In particular these cases may occur:

```
DCL a$ = 9 [<
FIELD = M, a [
  DCL b$ = 5 [<
  FIELD = M, b [
    DCL d#
    DCL e#
    DCL f$ = 3
  ] >]
  DCL c$ = 4
] >]
```

```
DCL a$ = 7 [<
FIELD = M, a [
  DCL b$ = 5
  DCL c$ = 4
]
FIELD = M, a [
  DCL d$ = 2
  DCL e$ = 2
  DCL f$ = 3
] >]
```

Figure 11. Containment relationship

Case 1 Exact size match.

Case 2 Redefinition uses less memory than the original variable.

Case 3 Redefinition uses more memory than the original variable.

The first case represents the ideal situation, perfect match between a variable and its redefinitions. This is the case of variable `b` and its redefinition as `d`, `e` and `f` in Figure 2. In this case we consider the redefinition finished when a variable is appended that makes the size of the content match the size of the container exactly.

In the second case, the sum of the size of variables within a redefinition is smaller than the original variable size. For instance, this would occur if `c` had size 3 instead of 4 in Figure 2. Technically, this requires to explicitly close the redefinition (*stopping condition*):

Case 2.1 The redefinition is explicitly closed by a `FIELD=M` statement, that resets the memory pointer to the next free available position.

Case 2.2 Another redefinition of the same variable starts before the full size is reached.

Case 2.3 A redefinition of another variable starts before the full size is reached.

Case 2.4 Declarations in the code are not enough to fit the size because the end of the variable declaration section is reached.

Case 3 occurs when the redefinition does not reach exactly the size of the original variable (case 1) and the redefinition is not terminated explicitly (case 2). In this case the redefinition is considered closed when a variable is added that crosses the boundary of the enclosing variable.

Since we do not know whether this structure corresponds to the actual intent of the developers, an error is reported and a manual fix intervention is requested. We recognize this instance as an explicit intention of the developer when one of the following stopping conditions appears immediately after the last variable in the redefinition:

Case 3.1 The redefinition is followed by the `FIELD=M` statement that resets the memory pointer.

Case 3.2 The redefinition is followed by another redefinition of the same variable.

Case 3.3 The redefinition is followed by a redefinition of another variable.

Case 3.4 There are no other declarations in the code because the end of the variable declaration section is reached.

An example of Case 3 is shown in Figure 10, where the declaration of `c` crosses the boundary of `a`. Since the declaration of `c` is immediately followed by a stopping condition (case 3.2), bracketing of the redefinition of `a` can be completed automatically, without requiring any user intervention. The declarations of `b` and `c` are put inside the square brackets for the `FIELD=M,a` instruction. For `d`, `e` and `f` in the example shown in Figure 10 the stopping condition that applies is number 3.4 (end of declaration section). The resulting bracketing is shown in Figure 11, right.

4 Experimental data

In this section, we first describe migration process and tools, as well as the system being migrated. Then, we report on some data that we collected when applying the proposed data model structuring techniques.

4.1 Migration process and tools

The legacy system contains two different kinds of data structures that deal respectively with *persistent* and *transient* data. Persistent data are stored on ISAM files, each containing one or more ISAM tables. The structure of most of the ISAM tables is described in a particular ISAM table (indeed, a meta-table) called the *dictionary*. This is a detailed description of the table meta-data, that includes not only type and size of table fields, but also supplementary information such as the fields used as discriminators as well as the discriminator values. Declarations for data coming from these ISAM tables are inside include-files that are periodically generated from the dictionary. When moving to Java, the dictionary must be translated as well, since its declarations have to be turned into class definitions that allow instantiating Java bean objects whenever a record is retrieved from the persistent storage.

A few remaining ISAM tables are described in developer maintained data structures, but not in the dictionary. All the other data structures contain transient data: they are used in the front-end interaction, they store intermediate results. The BAL code for transient data structure is manually maintained by the developers.

4.1.1 Dictionary

Considering the valuable information available in the dictionary, the analysis of persistent data structures is performed directly on it, instead of the generated include-files. The dictionary is converted by a pre-existing custom tool into an XMI representation that can be inspected with any XML library. We used XOM[2] (an XML manipulation library for Java) to analyze the dictionary representation and to generate the Java classes to access ISAM tables. The same cases, described in the previous section, apply both to data structures found in the user code and data structures documented in the dictionary. Hence, the same bracketing algorithm was used, but the implementation of the algorithm for the dictionary is based on Java/XOM.

4.1.2 User code

Analysis of the user code is performed in three main steps:

- **Code normalization**, the code is normalized in order to make the subsequent analysis and transformation simpler;
- **Fact extraction**, a number of facts is extracted from the code and used in final step;
- **Data structure inference**, the containment relationship is identified and all the possible overlays are grouped together.

In the first step the code is normalized and code ambiguities are resolved. We use agile parsing [6], modifying grammar and language to distinguish between ambiguous cases. For example, in BAL the same syntax (i.e., brackets) is used for array access and for function invocation. In the code normalization step, we change the declarations and all uses of arrays so as to comply with the C/Java syntax (square brackets). Unique naming [7] is used to generate identifiers that are unique within the system, regardless of their scope. Unique naming is required because segments can have local variables and global named constants can be used as sizes in segment local variables. Moreover, in case of reused variable names, local names hide global names.

In the first step we also identify and mark the portions of code originated from the expansion of include-files that are generated from the dictionary. The data structures in these portions of code are not analyzed in step 3, since their analysis is carried out directly on the dictionary where they come from.

In the second step (fact extraction) the code is analyzed and information about it is stored in a data base. The most important facts produced in this phase deal with type and length of all variables and constants. In this step, the combination of information from multiple files into a single data base allows us to resolve external information. An example is when the length of a variable is given by a constant or macro from another file.

The third step is the application of a source level transformation that searches for each of the cases described in the previous sections and inserts appropriate brackets to indicate the full extent of the boundaries of field redefinitions. In this step, size information is used to understand when different fields overlay in memory. Redefinitions of the same field are grouped together and moved next to the field declaration (square-angular bracketing), so that unions are immediately recognizable.

All three analysis steps for the user code have been implemented using the Txl language [4].

[2]http://www.xom.nu/

4.2 The legacy system

The system that we are migrating is a production banking application which supports all functionalities necessary to operate a bank, including account management, financial products management, front-desk operations, communications to central bank and other authorities, inter-bank communications, statistics and report generation. The user interface is character-oriented and the overall architecture is client-server, with the client operating mostly as a character terminal. The execution environment is a proprietary platform called B2U.

Lines of code (user code)	8,673,250
Lines of code (after expansion)	14,099,436
Number of source code files	2,701
Number of ISAM files	1,339
Number of ISAM tables	5,893
Number of unique ISAM tables	3,950

Table 1. Features of the system being migrated

Table 1 shows some indicators of the characteristics of the system being migrated. The application is quite large (around 8 MLOC). Since the BAL language admits preprocessor directives, the actual input to our analysis and transformation tools is the preprocessed (expanded) source code, with an approximate growth factor of 1.63. The persistent storage is also pretty large in terms of ISAM files and tables. For the latter, the correct number to consider is the number of unique ISAM tables. Some tables are just duplicates of other tables, having exactly the same structure. In such a case, only one table, representative of the entire equivalence class, is actually translated to Java.

4.3 Migration results

Case	User code	Dictionary
Case 1	425,988	8,279
Case 2	27,100	65
Case 3	13,850	4
Case 2.1	1,371	0
Case 2.2	13,121	36
Case 2.3	11,642	11
Case 2.4	966	18
Case 3.1	5,207	0
Case 3.2	1,645	1
Case 3.3	4,508	3
Case 3.4	2,490	0
Case err	5,286	45

Table 2. Occurrences of structuring cases

Table 2 shows the frequency of the cases considered during the inference of an object model from the existing flat memory model. The table is split into two columns, associated with data model inference for user code vs. dictionary. Case *err* occurs whenever none of the case-based heuristics applies and manual intervention is required. Manual fixes will be performed by the engineers who developed the original legacy system, they are BAL expert.

As apparent from Table 2, most of the cases, both in user code and dictionary, can be handled by the simplest of the cases in our case analysis: exact size match (Case 1). Case 2 (redefinition of less memory than declared) seems to prevail on Case 3 (redefinition of more memory than declared), both in user code and dictionary. Whenever a size mismatch occurs, the presence of a successive redefinition of the same or another variable can be exploited to infer the data structure boundaries in most situations (see Cases 2.2, 2.3, and 3.2, 3.3).

The 45 error cases remaining in the dictionary have been fixed by means of a tool that allows the user to specify the dictionary transformations required to fix the problems. The tool accepts instructions such as “change field length” or “insert container field before a given field”. By manually providing such instructions to the tool, we have been able to produce a version of the dictionary that can be processed completely automatically and from which it is possible to generate all Java classes required to represent the persistent data. This manual intervention required 5 working days (inclusive of dictionary fixing tool development). A similar code fixing tool is under development for the 5,286 error cases remaining in the user code. Even if the cases to solve are quite a lot, a further investigation showed that they are not independent, they all refer to just 442 recurring variables. In all other cases (466,938), Java classes have been automatically generated for the user code.

Table 3 shows the number of classes, interfaces and unions (counted also as classes) generated for user code and dictionary. Interfaces are generated only to support proper definition of unions in Java. Numbers indicate that in most of the cases, unions are not necessary, in that the given data structure is accessible through a single view. In the dictionary, the total number of unions is relatively small (263), so that it is reasonable to plan their complete manual elimination from the generated Java code. Among the classes generated from the dictionary, 88 satisfy the pattern shown in Figure 5 (mutually exclusive overlay). For these cases it is possible to take advantage of inheritance and to use a discriminator to decide which subclass to instantiate or downcast to.

	User code	Dictionary
Classes	510,108	12,402
Interfaces	148,621	753
Unions	29,394	263

Table 3. Generated Java code

Current BAL developers will maintain the translated code (after intensive Java training). Maintainability issues have been discussed with the customer. Decisions about the final system design and translation have been taken in concert with him.

5 Related work

The problem of migrating a legacy software system to a novel technology has been widely addressed in the literature by different approaches. The different strategies have been classified by [1] into (1) redevelopment from scratch; (2) wrapping; and, (3) migration. In their view, even the migration strategy requires substantial redevelopment. Our contribution belongs to the third class and consists of a set of automatic transformations.

Migration to object oriented programming and extraction of an OO data model from procedural code are the topics of several works [2, 5, 15, 16, 17]. Class fields originate from persistent data, user interface, files, records and function parameters, while class operations come from the segmentation of the program according to branch labels in the work by Sneed et al. [15]. Other works on object identification rely on the analysis of global data and of the code accessing them [2, 11, 13]. Since a record is too large and often contains unrelated data, cluster analysis was used [17] to identify groups of related fields within a record. In order to decide which data and which routines should be grouped together into classes, object-oriented design metrics (Chidamber and Kemerer) have been used to guide the migration [3, 5].

Type inference was used to acquire information about variables in legacy applications that goes beyond that conveyed by the declared type, so as to simplify migration toward a programming language with a richer and stronger type system [12, 14]. For instance, type inference was applied to Cobol [18, 19] to determine subtypes of existing types and to check for type equivalence. Static analysis and model checking have been used on Cobol to determine when a scalar type should be better regarded as a record type [10] and to determine unions the variants of which are consistently accessed through discriminators [8, 9].

The work presented in this paper differs from the existing literature in that it deals with a starting data model permitting arbitrary overlays in memory. Our work represents the first step – reverse engineering a structured data model – toward an OO model of the data.

6 Conclusions and future work

We have presented an algorithm for the reverse engineering of a structured data model from a data model based on arbitrary memory layouts. Although described in the context of a real, ongoing migration project, the proposed approach is quite general and applies to a number of programming languages that support arbitrary data layout in memory. For example, it would be relatively easy to apply the same technique to the structured assembly language of mainframes, in migration projects targeting a language with a structured data model (e.g., Cobol). Often, portions of IT systems on mainframes are written in assembly, and we are aware of at least one major system written entirely in assembly.

Our future work will be focused on the remaining issues that affect the migration of the data model. In particular, one important improvement of the migrated data model can be achieved if union discriminators are recognized even when not explicitly documented in the dictionary. We are developing a technique for this problem, along the lines of the existing literature on this topic [8, 9].

References

[1] J. Bisbal, D. Lawless, B. Wu, and J. Grimson. Legacy information systems: issues and directions. *Software, IEEE*, 16(5):103–111, September/October 1999.

[2] G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Software: Practice and Experience*, 26:25–48, January 1996.

[3] A. Cimitile, A. D. Lucia, G. A. D. Lucca, and A. R. Fasolino. Identifying objects in legacy systems using design metrics. *Journal of Systems and Software*, 44:199–211, January 1999.

[4] J. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, August 2006.

[5] A. De Lucia, G. Di Lucca, A. Fasolino, P. Guerra, and S. Petruzzelli. Migrating legacy systems towards object-oriented platforms. *Software Maintenance, 1997. Proc. of International Conference on*, pages 122–129, 1-3 October 1997.

[6] T. Dean, J. Cordy, A. Malton, and K. Schneider. Agile parsing in txl. *Journal of Automated Software Engineering*, 10(4):311–336, October 2003.

[7] X. Guo, J. R. Cordy, , and T. R. Dean. Unique renaming of java using source transformation. In *Source Code Analysis and Manipulation 2007, Proc. of the 3rd IEEE International Workshop on*, Amsterdam, The Netherlands, September 2003. IEEE Computer Society.

[8] R. Jhala, R. Majumdar, and R.-G. Xu. State of the union: Type inference via craig interpolation. In *Tools and Algorithms for the Construction and Analysis of Systems, 2007. Proc. of the 13th International Conference on*, pages 553–567, 2007.

[9] R. Komondoor and G. Ramalingam. Recovering data models via guarded dependences. *Reverse Engineering, 2007. Proc. of the 14th Working Conference on*, pages 110–119, 28-31 October 2007.

[10] R. Komondoor, G. Ramalingam, S. Chandra, and J. Field. Dependent types for program understanding. In *Tools and Algorithms for the Construction and Analysis of Systems 2005. Proc. of the International Conference on*, pages 157–173, 2005.

[11] S.-S. Liu and N. Wilde. Identifying objects in a conventional procedural language: an example of data design recovery. *Software Maintenance, 1990. Proc. of the International Conference on*, pages 266–271, 26-29 November 1990.

[12] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. *Software Engineering, 1997. Proc. of the 19th International Conference on*, pages 338–348, 17-23 May 1997.

[13] S. Pidaparthi and G. Cysewski. Case study in migration to object-oriented system structure using design transformation methods. *Software Maintenance and Reengineering, 1997. Proc. of the first Euromicro Conference on*, pages 128–135, 17-19 March 1997.

[14] G. Ramalingam, R. Komondoor, J. Field, and S. Sinha. Semantics-based reverse engineering of object-oriented data models. In *Software engineering, 2006. Proc. of International Conference on*, pages 192–201, 2006.

[15] H. Sneed and E. Nyary. Extracting object-oriented specification from procedurally oriented programs. *Reverse Engineering, 1995. Proc. of the 2nd Working Conference on*, pages 217–226, 14-16 July 1995.

[16] H. B. K. Tan and T. W. Ling. Recovery of object-oriented design from existing data-intensive business programs. *Information and Software Technology*, 37:67–77, 1995.

[17] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. *Software Engineering, 1999. Proc. of International Conference on*, pages 246–255, 1999.

[18] A. van Deursen and L. Moonen. Understanding cobol systems using inferred types. *Program Comprehension, 1999. Proc. of the Seventh International Workshop on*, pages 74–81, 1999.

[19] A. van Deursen and L. Moonen. Exploring legacy systems using types. In *Reverse Engineering. Proc. of the seventh Working Conference on*, pages 32–41. IEEE Computer Society Press, 2000.