Eighth IEEE International Working Conference on Source Code Analysis and Manipulation

Using program transformations to add structure to a legacy data model

Mariano Ceccato[(1)], Thomas Roy Dean[(2)], Paolo Tonella[(1)]
(1) FBK-irst, Trento, Italy
(2) Queen's University, Kingston, Canada
ceccato@fbk.eu, tom.dean@queensu.ca, tonella@fbk.eu

Abstract

An appropriate translation of the data model is central to any language migration effort. Finding a mapping between original and target data models may be challenging for legacy languages (e.g., Assembly) which lack a structured data model and rely instead on explicit programmer control of the overlay of variables.

Before legacy applications written in languages with an unstructured data model can be migrated to modern languages, a structured data model must be inferred. This paper describes a set of source transformations used to create such a model as part of a migration of eight million lines of code to Java. The original application is written in a proprietary language supporting variable layout by memory relocation.

1 Introduction

Adaptive maintenance is defined as a change to software to enable it to work in a changed environment [18]. It may be due to several reasons, such as the obsolescence of an old technology or language. Domain specific, proprietary programming languages represent a typical instance of this problem. They may become a major impediment to future developments and innovation, especially when they are tied to a proprietary development, compilation and execution environment. In such cases, supporting the language and the associated production and execution environment may require even more resources than those dedicated to the core business of the software company. Moreover, customers may become more technology-aware, hence requesting details about the underlying technology and expressing concerns about the usage of non-standard solutions, having no consolidated support and limited user community. This is becoming especially true in the days of the open source software development. In such a context, migration to a modern platform and to a modern programming language is an appealing option.

Central to any migration effort is the translation of the data model. Decisions made about the representation of the data model will have strong implications for the rest of the translation. While object oriented languages are centered around an object data model, which takes advantage of constructs such as inheritance and composition, legacy languages often enforce a functional decomposition of the code, which is orthogonal to the underlying data model. The latter is often based on the data organization offered by the persistency layer, properly represented in the language constructs. This paper describes the use of source transformations to infer structure in the data definitions of a legacy, proprietary language in preparation for an automated migration to a modern Object-Oriented language.

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 as well as unstructured control statements (e.g. GOTO). Calls between programs are supported, and a preprocessor provides the programmer with the ability to isolate common code that may be included in more than one program. BAL programs are complied to a byte code representation and run on a virtual machine, providing a great deal of portability. This portability is one of the main reasons for the choice of Java as the target platform (a decision made by the client).

We bridged the gap between the BAL data model and the Java data model by defining a set of source code transformations that take advantage of idiomatic usages of the BAL data model. Such uses have been discovered by code inspections, in an iterative data structuring process, consisting of: (1) definition of data structuring rules (heuristics); (2) application of the rules to the code base; (3) inspection and/or automated analysis of the cases not yet handled by the structuring rules. The resulting set of transformations allowed us to map the BAL data structures in the original application (8 MLOC) into a set of Java classes.

The rest of the paper is presented in 5 sections. Section 2 gives a short introduction to the BAL data model. Section 3 discusses the transformations in more detail. Related work is presented in Section 4, followed by conclusions.

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.9

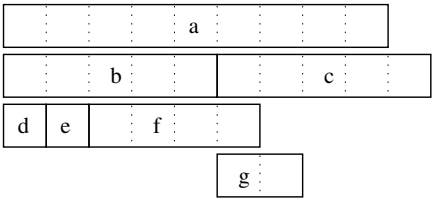
2 Data model

An appropriate translation of the data model is central to any migration effort, and our project is no exception. This section gives a short overview of the BAL data model. The data model is discussed in more depth in a separate report [3].

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 single and two byte signed integer values. Variables of the BCD and string data types can be of various lengths, and the developer must specify the length in bytes if she wants some thing different than the default length (eight and sixteen bytes respectively). Unlike languages such as C, there is no dynamic allocation and the length of all variables is known at compile time.

The BAL data model is based on the notion of memory relocation. Each newly declared variable can be either allocated on the next available region in memory, or it can be a relocation of a previously defined variable. The relocation data model allows for arbitrary aliasing among variables as well as for arbitrary definition of multiple views (unions) insisting on the same underlying memory region (i.e., sequence of bytes). Such views are not constrained to be within the boundaries of the relocated region. On the contrary, they can span multiple memory regions that were previously regarded as separate data structures.

The data relocation is accomplished with the `FIELD=M,VAR` statement, which is analogous to the *`.=variable address`* directive available in most assembly languages. An example is shown in Figure 1. Indentation is used in the example to indicate programmer intention, but is not mandatory in the language. The variable `a` is a string variable (indicated by a type specifier of '$') and is nine bytes long. The following `FIELD` statement resets the current variable position (i.e. the memory location of the next declared variable) to the beginning of the variable `a`. As a result, the variable `b`, a string variable of length five, occupies the same five bytes as the first five bytes of `a`. The variable `c`, also a string variable, takes the next five bytes, going one byte beyond the memory allocated for `a`. Thus all nine bytes of the variable `a` are shared with other variables, but `c` as an extra byte of its own. The variable `b` is further redefined by the variables `d`, `e`, and `f`. The variables `d` and `e` are byte variables (the type specifier '#') while `f` is another string variable. Since `f` is four bytes long, its last byte overlaps with the first byte of `c` as well as the sixth byte of `a`. Finally, `g` is declared starting from the beginning of `c`. Being a short ('%' type specifier), it occupies two bytes. Variants of the declarations shown in Figure 1 may include one or two dimensional arrays of any of the four primitive types. The `FIELD=M` that concludes these declarations refers to no variable. Its meaning is that the next declared variable will be allocated starting from the first free location in memory (i.e., immediately past the end of `c`).




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

Figure 1. Variable declarations in BAL

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 second is that it is the developers' responsibility to ensure that the sizes of all of the variables are correct. For example, in Figure 1, if the variable `a` is intended to be a reference to the entire record, its size is wrong (it should be 10). The third 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.

In fact, there are several issues in the code in the figure. The variable `a` is in fact shorter than the subfields `b`, `c`. When this happens in the real code, it may be unclear if it is an error, or represents an alternate shorter view to the same data. Similarly for `b`: if `b` is intended to be a container for the subfields `d`, `e`, `f`, its size should be increased to 6. The other issue is the location of the redefinition of `b`. In a language such as COBOL or C, the variables `d`, `e`, and `f` would normally be declared as part of the aggregate field that they form. In BAL they could be located anywhere within the same scope of declarations (i.e. global or local).

The developers usually keep the definitions of complex data structures in separate files which are included by the

preprocessor. One special case is that of the data structures used to access persistent ISAM (Indexed Sequential Access Method – a data file format common in legacy systems) tables. These record definitions are stored in a data dictionary and contain full hierarchical storage information. Include files containing BAL declarations are generated from these dictionary entries, losing some of the hierarchical information in the process. Since the information exists in the dictionary, these files are treated specially by our migration process as the Java classes can be generated from the dictionary directly. One wrinkle is that the data dictionary contains a few size mismatches [3] and as a result, so do the generated include files. Such size mismatches are basically similar to those shown in the example in Figure 1. While not affecting the correctness of the final code, these problems complicate the process of recovering the data model and sometimes require manual fixes.

The data structures for other temporary data files and internal storage do not have entries in a dictionary and the structure of these records must be inferred solely from the BAL code. However, when analyzing any structures that are local to the program, we have considerably more flexibility in how the fields can be reorganized in order to infer reasonable data structures.

Given this unstructured legacy data model, the next section presents a sequence of transforms which group the appropriate fields together, moving variables as necessary.

3 Approach

The recovery of a structured model from a flat, unstructured data model is part of a larger project involving the migration of the legacy code base from BAL to Java, a somewhat complex process. At the core of data structure recovery is one particular phase which we call *bracketing* and which precedes the rest of the migration (e.g., translation of the statements). However, several other phases of the migration process precede bracketing, providing several generic transformations that are of use to the entire process including bracketing. We present some of these transformations, although we limit the discussion to those elements that are used by bracketing directly. These transformations include a normalization of the code and the extraction of information about the programs for use in later transforms. TXL [5] is used for most of the transformations, including bracketing itself.

3.1 Normalization

The normalization phase is the gateway to the entire migration process and performs several transformations designed to make the rest of the process easier. The normalization transforms related to bracketing are preprocessing, unique naming and persistent file identification.

The BAL compiler produces a partially preprocessed file during the compilation process. In producing this file, the BAL compiler resolves all inclusion statements as well as any conditional compilation. Each of the lines of the file end with an annotation that gives the line number and name of the original source file (e.g. the name of the include file that contains the statement). Macro definitions and comments have been removed, although the use of macros in the code has not been expanded. Since the macros will be translated separately to Java constants and methods, the presence of the original names in the code will allow us to use them in the final translation.

The first set of transformations is to uniquely name [8] all of the identifiers in the programs. Unique identifiers are strings, which contain the names of each of the parent scopes of the identifier. So the global variable `A` in program `P` has the unique identifier `"P A"`, while the variable `B` in the function `C` in the program `P` has the unique name `"P C B"`. Since program names are unique, and functions and variables may not have the same name, this approach gives a globally unique name to all identifiers in the system. We use agile parsing [7] techniques to extend the grammar to include the unique identifiers as annotations on all occurrences of the identifiers (definitions and uses). These annotations take the form:

```
["UID" # identifier]
```

Since BAL has simpler naming rules than Java, the unique naming process is also simpler, requiring only the resolution of macro names from include files (since the instances of the macros are not expanded) and the external functions defined in libraries. For clarity and ease of reading, the examples in this paper do not show the unique naming annotations.

As mentioned earlier, some of the include files are generated from the data dictionary. Since some extra structural information is present in the data dictionary (and lost when the BAL include files are generated), it makes sense to generate the Java classes from the dictionary. However, we must still recognize where such data structures are used in the code so that instances of classes may be generated during migration. They also complicate the bracketing process since they may also be included as a subrecord of a larger record. Thus the last transform of the normalization process is to use the annotations inserted by the compiler at the end of each line to group all of the fields for a given persistent file structure into a single unit. In this way, a single object allocation can be generated in Java, replacing the entire persistent data structure included.

3.2 Fact extraction

Facts in RSF format [23] are extracted from the source code as part of the migration process. These include facts about the type of variables, their size, the structure of the BAL source and the names of externally visible entities. Facts are triples of the form:

```
factName uid uid_or_value
```

The UIDs created by unique naming during normalization form a link between the facts and the source code, and allow the facts to be read and used by transforms later in the migration process. The main set of facts relevant to the bracketing process are the size facts. In fact, the layout of variables in memory depends on where they are allocated (i.e., whether they are relocations of other variables or not) and on their size.

The size fact gives the size of each variable and constant in the code. This is accomplished by first extracting constant declarations from the code. The transform then extracts all variables and their associated size expressions, inserting default sizes were appropriate. Since named constants may be used when specifying the size of variables, a constant propagation algorithm is run on the size expressions to determine the size of each variable.

3.3 Bracketing Process

Bracketing is the core transform used to add structure to the legacy BAL data model. The bracketing algorithm consists of finding out what is the full extent of a `FIELD=M` and in enclosing it between square brackets. It starts at a `FIELD=M` statements and it contains one or more declarations (`DCL`), possibly with nested `FIELD=M`. In addition to the square brackets for the `FIELD=M` instruction, we need a second kind of brackets used to group together all the redefinitions of the same declaration (`DCL`). We call this second kind of brackets square-angle brackets (i.e. `[ < ... > ]`). They start at a declaration and enclose all `FIELD=M,var` statements that insist on the same declared variable. When a `DCL` has no `FIELD=M` statement insisting on it, the square-angular brackets contain an empty sequence. Hence they are omitted.

During the translation to Java, square-brackets and square-angular brackets play different roles. Square brackets identify the boundaries of classes, with nested `FIELD=M` representing cases of the composition relationship between classes. Square-angular brackets discriminate between classes and unions. When more than one `FIELD=M` insists on the same variable declaration, a union must be generated instead of a class.

Since unions are not available in Java, we resort to an equivalent code organization, which implements the copy on read/write protocol. The generated class implements a set of interfaces, associated with the different union variants, and declares a set of data members, each referencing one union variant. At each point in time during execution, at most one variant is active (i.e., at most one data member is non-null). When the code switches from one view to another one (e.g., it sets an attribute available in one view and then it reads an attribute available in another view) the active variant is changed and data are copied from the previous active variant to the new one.

```
1 Delimit persistent data definitions
2 Fix size mismatches in persistent data
3 Fix size mismatches in formal parameters
4 Bracket all FIELD=M
  4.1 Initialize size
  4.2 Initialize brackets
  4.3 Repeat
     4.3.1 Repeat
          4.3.1.1 Fold declarations
          4.3.1.2 Fold constants
          4.3.1.3 Fold redefinitions
     Until no fold rule applies
     4.3.2 Fold persistent data
  Until no fold rule applies
5 Report errors
6 Move redefinitions
```

Figure 2. Bracketing algorithm

The algorithm used for bracketing is shown in Figure 2. We discuss each element of the algorithms separately.

3.3.1 Persistent Data

The first step of the bracketing algorithm consists of the identification of the persistent data definitions (coming from the dictionary), which are delimited and separated from the user data definitions, within the preprocessed files.

A single persistent data definition file corresponds to the definition of an ISAM file which, in turn, can contain more than one table. During normalization, each included file is delimited as a whole by inserting proper code annotations. However, the included file may contain more than one persistent data structure each associated with one table declared in the file. In the original, unpreprocessed BAL source files, each table definition is delimited by its own preprocessor macro, giving the programmer fine grain control over the inclusion of table definitions in the code. In particular, each table definition is enclosed between `#ifdef table_name` and `#endif` preprocessor lines. Figure 3 shows an example of these files. The file `a.bal` (Figure 3(a)) includes the generated include file `tabx.bal` which in turn contains three tables

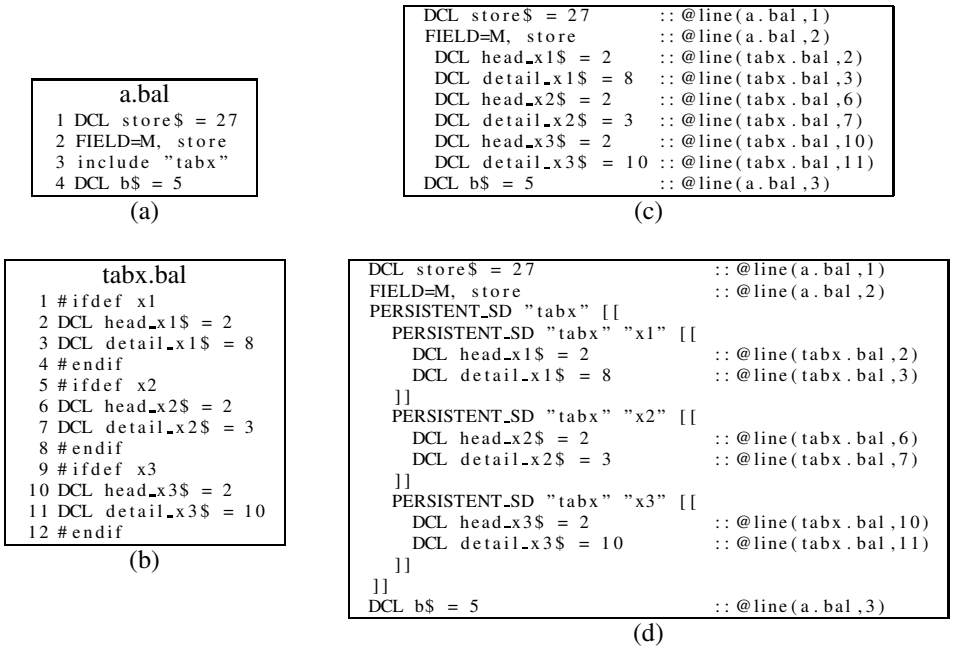


Figure 3. Delimiting persistent data definitions: program (a) and include file (b), preprocessed file with expanded include (c) and marked code (d)

(`x1`, `x2` and `x3` (Figure 3(b)). We use a program that analyzes the files generated from the dictionary to determine that the extents of these tables are (1:4), (5:8) and (9:12) respectively. Figure 3(c) shows the preprocessed version of `a.bal`, and in particular the annotations appended to each line by the preprocessor. Figure 3(d) shows the results of both the identification of the persistent include file (bracketed by `PERSISTENT_SD "tabx" [[ ... ]]`), and the identification of the tables within the file (bracketed by `PERSISTENT_SD "tabx" "tablename" [[ ... ]]`). The remaining figures in this paper will omit the inclusion annotations to simplify the presentation of the transforms.

3.3.2 Persistent Data Size

As mentioned previously, persistent data structures contain a few size mismatches. In fact, the BAL data model is quite robust with respect to such mismatches, which usually have no impact on the correctness of the program execution. In fact, the pointer to the next free location in memory can be easily reset to a legal position through `FIELD=M` with no parameter. Moreover, container size mismatches are usually not an issue. Provided the container has all the information needed by the computation based on it, its size can be larger or shorter than the containees without any visible effect. However, from the point of view of recovering a structured data model, size mismatches represent a major impediment to proper bracketing.

In order to fix the known size mismatches, a table of fixes is read in the next step (Step 2 in Figure 2) and then propagated to the definitions of the variables bracketed by the previous step. Examples of such corrections are changing the size of a field (usually increasing it to include all contained fields) or inserting a missing container for a sequence of fields. The table of fixes was produced manually, addressing the warnings raised by a size checker developed as part of the bracketing tool. The fixes have been validated by the programmers.

3.3.3 Variable Parameter Size

The third step addresses the formal parameters of functions. When a variable is passed by reference, the compiler does not enforce that the type in the signature of the function corresponds to the type of the actual argument, and often the size and types do not match. The particular pattern we have identified as occurring most often is that programmers declare the formal parameter as a BCD without size information, hence, it is 8 bytes long, by default. However, since the parameter is passed by reference, its actual location in memory starts exactly where the actual argument is located. Just as global and local variables may be redefined, so may parameters. In particular when a variable parameter is rede-

fined with a FIELD=M statement, all successive variables are defined relative to the actual argument. The only way to reset the storage to the memory space for local variables is to use a FIELD=M statement that relocates another local variable, or a FIELD=M statement without a variable, which resets the allocation pointer to the next available location in local variable memory. Thus the transform for identifying the structure of a pass by reference parameter is slightly different than that for global or local variables or for pass by value parameters.

We use the information provided by the redefinitions to correct the function signature and indicate the correct size of each parameter that is passed by reference. On some occasions, more than one redefinition on the same parameter specify different sizes. In this case, the maximum among the candidates is used for the correction. For example in Figure 4 function `f1` declares two parameters, the first (i.e., `p1`) is passed by value, while the second (i.e., `buffer`) is passed by reference (using the keyword `VAR`). Even if the size of `buffer` is the default (16 bytes), we see by inspection that it is redefined twice with two different lengths (500 and 1000 bytes). Thus in this case, we assume the maximum size (1000 bytes).

```
FUNCTION f1 (DCL p1$ =1, VAR buffer)
   FIELD=M, buffer
      DCL a$ = 500
   FIELD=M
   ...
   FIELD=M, buffer
      DCL b$ = 1000
   FIELD=M
   ...
ENDF
```

Figure 4. Formal parameter size

3.3.4 Bracketing

Once annotations and fixes are done, the actual *bracketing* can start. The bracketing step is composed of four substeps. First, size information is read from the fact base and an in-memory data-base is initialized. As explained previously, the unique identifiers provide the link between the size facts and the declarations they represent. We have extended the base grammar of the BAL language to include our bracketing syntax as shown in Figure 5. The first definition adds the square bracketing syntax to the `FIELD` statement. The grammar for the square bracketing includes a number. This is where the transform stores the amount of memory that is left in the redefinition (initially the size of the variable that is being redefined). The second grammar definition adds the optional redefinition (square-angle bracketing) to variable declarations. A redefinition of the `bal_declaration` non-terminal adds both of these non-terminals as possible definitions.

```
define bracket_field
  [bal_field_stm] [opt '-] [number] '[
    [repeat bal_declaration]
  ']
end define

define union_bal_dcl_list
   [dcl_or_parameter] [bal_var_decl] '[' <
       [repeat bal_declaration]
    '>']
end define
```

Figure 5. Bracketing Grammar Extensions

The bracketing transform (Steps 4.1 and 4.2 in Figure 2) starts by adding an empty extent to all `FIELD` redefinition. The size field of the extent is initialized to the size of the variable that is overlaid. Figure 6 shows the result of this transformation. Empty square brackets have been added to the redefinitions of `a` and `b`. The initial sizes for these extensions are 9 for `a` and 5 for `b`.

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

(Original)

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

(Initialized)

Figure 6. Bracketing initialization

The transform proceeds by checking if the next entity following a bracketed field can be moved inside the brackets. This is achieved by comparing the size of the entity with the available space left in the field. Depending on the type of the entity that follows the bracketed extent, a different transformation applies. The possible transformations are Steps 4.3.1.1, 4.3.1.2, 4.3.1.3, 4.3.2 from Figure 2.

Fold declaration (Step 4.3.1.1). In case the entity to fold is a variable declaration (`DCL`), the transform folds in the variable if there is space left in the container. Figure 7 shows the results of this transform. In Figure 7(a), the first declaration following each of the square bracketed extents from Figure 6 is folded. In particular, the variable `b`, whose size is 5, is folded into the redefinition of `a`, reducing the size of that redefinition from 9 to 4. The variable `d`, whose size is 1, is folded into the redefinition of `b` reducing the available space from 5 to 4.

This transformation is applied until no other variable declarations can be folded into the appropriate overlay. After folding declarations `e` and `f` into the redefinition of `b`,

no other declaration folding is possible, the result of which is shown in (Figure 7(c)).

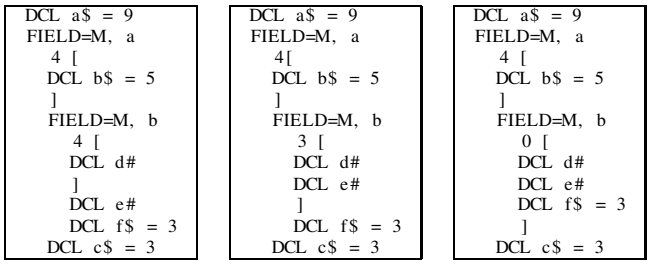


Figure 7. Folding declarations

Fold constants (Step 4.3.1.2). This is a special case of folding declarations which applies when the currently bracketed extent is followed by the declaration of a constant. Depending on the type of the constant, some space can be required in the variable pool. Tests revealed that byte and short constants do not require space and are substituted into expressions by the compiler at compile time. However BCD and string constants are allocated on the variable pool and they interact with the other (non-constant) variables. As an aside, this means that they are not really constants, and the values can be changed by assigning to an alias created by an overlay. Thus when folding a constant into a bracketed redefinition, the available space must be updated when a BCD or string constant is folded. In Figure 8 only constant strings *x1* and *x2* consume space when bracketed (3 bytes each), while *x2* and *x3* (byte and short constants) do not impact the available space.

Fold redefinitions (Step 4.3.1.3). The last case is where the item following a redefinition is another redefinition (i.e. another `FIELD` statement). A redefinition may only be folded, or nested, when:

1. The redefinition is fully bracketed. That is, the remaining space is zero; and,
2. The redefinition applies to a field already present in the bracketed extent.

In Figure 9(a), the redefinition of `b` can be folded into the redefinition of `a`. When redefinitions are folded, the available space is not updated, since they apply to space that has already been accounted for. Thus in the figure, the available space of `a` is not updated, because the redefinition of `b` does not actually consume space within `a`. In fact, the redefinition of `b` takes exactly the amount of space already reserved for field `b` (i.e., 5 bytes). After this transformation, the fist rule (*Fold declarations*, Step 4.3.1.1) re-applies and the last declaration is finally folded into `a` and the available space is reduced to 1 (Figure9(b)).

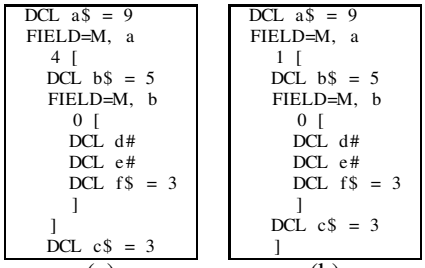


Figure 9. Folding redefinitions

The last case (*Fold persistent structure*, Step 4.3.2) applies when the current redefinition is followed by a persistent data definition. In this case, the persistent data definition can not be broken into parts, because it describes the structure of an entire ISAM table. Either it is moved entirely in the extent of the redefinition or it is left outside of it. The size of a table definition is computed as the sum of the fields contained in its definition. In order to get this calculation right, overlay extents must have been already computed for the whole table, otherwise the same variable space could be counted multiple times (redefinitions do not require new memory space). This is the reason for applying this transformation only after all the others have been successfully applied.

After folding the persistent data structures, repeated iterations of folding declarations, constants and redefinitions may be required to complete the transformation.

In the ideal situation, a perfect match should be found between the size of the container field and the size of its overlay. In such a case, the folding procedure stops when the available space of the overlaid variable reaches zero. However, since the BAL compiler does not enforce any size control among overlays, size mismatches could happen, so mismatches must be handled by the bracketing algorithm.

The first case of mismatch occurs when the size of the redefinition is smaller than the available space. Usually, in this case an explicit stopping condition is inserted by the developer. Stopping conditions can be:

- The redefinition is explicitly closed by a `FIELD=M` statement, that resets the memory pointer to the next free available position.
- Another redefinition of the same field starts before the full size is reached.
- A redefinition of another field starts before the full size is reached.
- Declarations in the code are not enough to fit the size because the end of the field declaration section is reached.

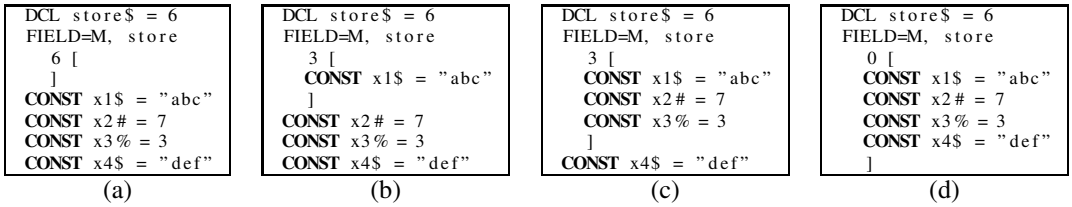


Figure 8. Folding constants

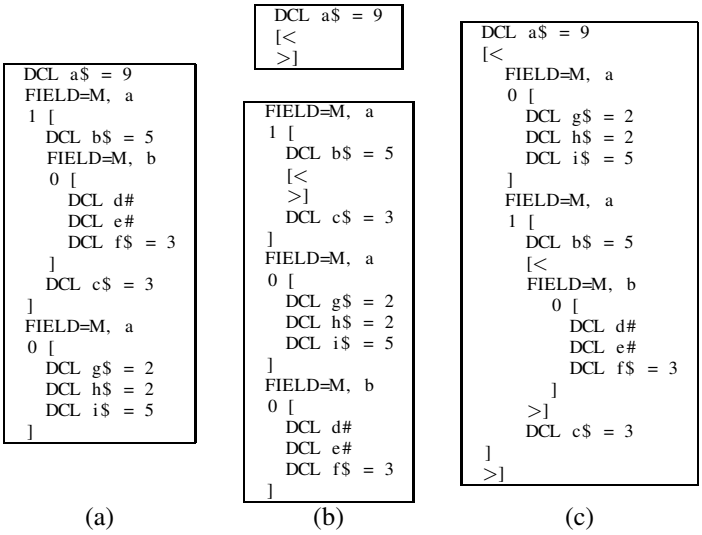


Figure 10. Moving transformation: (a) result of bracketing, (b) initialization, (c) move redefinitions

If neither a stopping condition is found nor an exact size match is reached, the second case of mismatch occurs: the redefinition is greater than the available space. In this case, the last entity to fold in the redefinition extent is the one that causes the extent to exceed the available space. We considered this a deliberate intention of the developer when the entity that crosses the boundary is followed by one of the previous stopping conditions. If no stopping condition occurs, an error is raised and manual intervention is required (Step 5, *Report errors*) to fix the size mismatch instance.

3.3.5 Move Definitions

Assuming that the bracketing steps have been completed and that any size mismatch has been manually fixed, the algorithm ends by moving redefinitions next to the declarations of the variables for which they apply (Step 6 in Figure 2, *Move redefinitions*). This is done in three steps.

1. Redefinitions are removed from the code and put in a separate data structure.
2. From the redefinitions, a list is created with the names of all the variables for which an overlay has been found. The declarations of variables in this list are initialized with an empty square-angle brackets.
3. Redefinitions are moved into the square-angular brackets of the variable for which they provide an overlay.

When this step is over, every field declaration is followed (in square-angular brackets) by all its possible alternative overlays. If there are more than one overlay, a union is recognized.

Figure 10 shows an example of results produced by the moving step. Figure 10(a) shows the output of the bracketing transformation for fields `a` and `b`. In Figure 10(b) the three redefinitions are removed and the declarations of `a` and `b` are initialized with empty square-angle brackets. Finally, redefinitions are moved under the proper declaration (see Figure 10(c)). It can be noticed that variable `b` has only one overlay, so it would be naturally translated into a normal Java class, while variable `a` has two alternatives redefinitions, thus it corresponds to a union.

The algorithm described in Figure 2 has been successfully applied to the legacy system being migrated (8 MLOC). Specifically, we have been able to add structure to

510,108 variable declarations for which one or more overlays were defined in the original code. Correspondingly, 510,108 Java classes have been generated. Among them, 29,394 are unions (i.e., contain multiple variants, deriving from multiple alternative overlays). The stopping conditions used to automatically handle the occurrence of size mismatches were pretty effective. They allowed us to solve automatically 81,900 cases, leaving only a few size mismatches to be fixed manually. The effort involved in such manual intervention accounted for less than a working week of a person.

3.4 Optimizations

Our first simplistic implementation of the square bracketing transform took almost 7 days to execute over the entire BAL codebase (8 MLOC). Two simple optimizations were applied. The first of these was an application of agile parsing. The non-terminal that represents a declaration was split into two non-terminals, one for foldable declarations (variable declarations, constant declarations and field statements with variable redefinitions) and one for declarations that are not foldable (field statements without variable declarations). This allows the parser to provide the first categorization and allows a simple pattern match in TXL to guard the rule.

The second optimization was to stratify the bracketing rules to a one pass rule that identifies potential folding locations using the grammar change described above, checks that space remains in the extent and then invokes a subrule which uses the TXL skipping statement to limit the rule to folding only at that location. A grand parent rule applies the one pass rule until no further changes are made. The result of these two optimizations reduced the transformation time of the entire BAL code base to less than six hours (from 7 days), a significant improvement.

4 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 object oriented data model from procedural code are the topics of several works (e.g., [2, 6, 17, 19, 20]). 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 migration of legacy procedural code to an object-oriented design described by Sneed et al. [17]. For similar purposes, data flow analysis and the classification of data elements into constants, user inputs/outputs and database records are used in the *augmented object model* by Tan et al. [19]. Sneed [16] migrated Cobol code to Object-Oriented Cobol.

Other works on object identification rely on the analysis of global data and of the code accessing them [2, 12, 14]. Since a record is too large and often contains unrelated data, cluster analysis was used [20] to identify groups of related fields within a record. Concept analysis is then applied to group together data and functionalities into candidate classes. In order to decide which data and which routines should be grouped together into classes, object-oriented design metrics (Chidamber and Kemerer) are also used to guide the migration [4, 6], so as to avoid a poor design quality in the resulting system that would pose maintenance problems. Classes are still based on persistent data stores and routines are assigned to classes, such that the final result minimizes the object coupling metric.

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 [13, 15]. For instance, type inference was applied to Cobol [21, 22] 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 [11] and to determine unions the variants of which are consistently accessed through discriminators [9, 10].

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. This requires a specific inference technique, that takes into account size and offest information explicitly. This paper is the companion of a Technical Report [3], where the target data model and the migration path from the legacy to the new data model are described in detail. On the contrary, in the present paper, we do not provide details about the target data model or the migration process. We focus instead on the automated program transformations that have been defined to actually implement the migration strategy described in the companion Technical Report [3]. The interested reader can refer to the Technical Report to get the full picture of the data model migration process being followed.

5 Conclusions and future work

We have described a set of transformations, based on the notion of declaration folding, that can be used to add structure to an unstructured data model which supports arbitrary variable overlays in memory. Such transformations

are complemented by a set of heuristics (stopping conditions) used to decide how to manage automatically cases of size mismatches for which a reasonable structure can be superimposed automatically. We believe this approach to be quite general and applicable to any programming language supporting memory relocation and arbitrary variable layout. This is the case, for instance, of Structured Assembly, a programming language still in use with applications running on mainframes.

The proposed algorithm allowed us to structure automatically approximately half million variables with overlays and to determine which of them correspond to unions (around 30 thousands). The manual intervention required to fix the size mismatch cases falling outside the proposed heuristics was quite limited (a few person days in total). The machine time for the transformation was optimized, thanks to careful crafting of the transformations in TXL.

In our future work, we will complete the migration from BAL to Java, addressing the problems associated with the translation of the program structure and of the statements (which involves also `GOTO` elimination). With regards to the migration of the data model, we will investigate the possibility of inferring inheritance relationships among classes, based on the identification of memory overlays that redefine with specialization (e.g., adding more variable declarations) existing data structures.

References

[1] J. Bisbal, D. Lawless, B. Wu, and J. Grimson. Legacy information systems: issues and directions. *Software, IEEE*, 16(5):103–111, Sep/Oct 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] M. Ceccato, T. R. Dean, P. Tonella, and D. Marchignoli. Inference of a structured data model in migrating a legacy system to Java. Technical report, FBK-irst, Trento, Italy, April 2008.

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

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

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

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

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

[9] R. Jhala, R. Majumdar, and R.-G. Xu. State of the union: Type inference via craig interpolation. In *TACAS*, pages 553–567, 2007.

[10] R. Komondoor and G. Ramalingam. Recovering data models via guarded dependences. In *WCRE*, pages 110–119, 2007.

[11] R. Komondoor, G. Ramalingam, S. Chandra, and J. Field. Dependent types for program understanding. In *TACAS*, pages 157–173, 2005.

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

[13] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *ICSE*, pages 338–348, 1997.

[14] S. Pidaparthi and G. Cysewski. Case study in migration to object-oriented system structure using design transformation methods. *Software Maintenance and Reengineering, 1997. EUROMICRO 97., First Euromicro Conference on*, pages 128–135, 17-19 Mar 1997.

[15] G. Ramalingam, R. Komondoor, J. Field, and S. Sinha. Semantics-based reverse engineering of object-oriented data models. In *ICSE*, pages 192–201, 2006.

[16] H. Sneed. Migration of procedurally oriented cobol programs in an object-oriented architecture. *Software Maintenance, 1992. Proceerdings., Conference on*, pages 105–116, 9-12 Nov 1992.

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

[18] E. B. Swanson. The dimensions of maintenance. In *ICSE*, pages 492–497, 1976.

[19] 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.

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

[21] A. van Deursen and L. Moonen. Understanding cobol systems using inferred types. In *IWPC*, pages 74–, 1999.

[22] A. van Deursen and L. Moonen. Exploring legacy systems using types. In *WCRE*, pages 32–41, 2000.

[23] K. Wong. *The Rigi User's Manual – Version 5.4.4*. June 1998.