Reverse-Engineering COBOL via Formal Methods

K. Lano
P.T. Breuer
H. Haughton
J.Estdale

2487-TN-PRG-1049  Version 1.6
May 1991

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford  OX1 3QD
England

Copyright © 2013  K. Lano
Classification and Approval

Classification: Internal to ESPRIT REDO project no. 2487

This document is for use only with Project P2487 and may not be used for any other purpose unless specifically authorised in writing by the Project Manager, having obtained the agreement of the Project Management Committee.

Approved for issue as per above classification:

Author(s): K. Lano  
P.T. Breuer  
H. Haughton  
J. Estdale

Approved: WP Leader

Date: May 1991

Doc. Id.: 2487-TN-PRG-1049 Version 1.6

Doc. type: Technical Note

Status: Final

Work Package: 2

Task: 215

Distribution: All REDO project partners

Confidentiality: Within REDO

Document History
Version 0.8 – First issued January 1991
Version 1.0 – February 10th 1991
Version 1.5 – May 16th 1991 Version 1.6 – May 31st 1991

Document Purpose
To Describe Formally based reverse-engineering Techniques for COBOL

Acknowledgement

This report reflects work which is funded or partially funded by the Commission of the European Communities (CEC) under the ESPRIT II programme project no. 2487: “REDO: REengineering, validation and DOccumentation of systems”
Reverse-Engineering COBOL via Formal Methods

K. Lano   P.T. Breuer*   H. Haughton   J.Estdale†

May 1991

Abstract

We describe methods and software tools which aid in reverse-engineering COBOL application programs back to the specification stage (and in validating them against the specification). The aim is to create object-oriented abstractions from the implementation which capture the design concepts accurately, and the central process which the tools support is ‘transformation from formalism to formalism’, first from COBOL to the intermediate language UNIFORM, then from UNIFORM to a functional description language, and then to the specification language Z. In the process data-flow diagrams, entity-relationship diagrams and call-graph, and other types of information, are extracted from the code.

Introduction

The COBOL language hinders program comprehension in several ways: by its use of formats instead of types (which means that each assignment between variables usually involves an implicit conversion of data from one format to another), by the mixed use of PERFORM statements and the ordinary fall-through execution of paragraphs, by the inclusion of unstructured constructs such as GO TO’s and ALTER GO TO’s, and by excessive proliferation of specialised verbs and variants of verbs, the exact semantics of which requires very specialised knowledge on the part of the programmer.

All these features make an organised programme for going about reverse engineering COBOL applications a necessity, and in the following sections we will set out just how to go about it using the tools which we have developed at Oxford as part of the REDO project† (of course it is possible to do what the tools do without using the tools themselves).

Picking up clues from COBOL

On the positive side, COBOL data declarations provide significant information about the structuring of the data that the program is expected to manipulate: they bind record names and field names to files, and specific index variables to arrays. This means that one can tell which file a record belongs to from its name alone. COBOL therefore already contains some rudimentary object-orientated features, and we seek to enhance these as reverse-engineering proceeds.

Moreover, in addition to the expected type and value bindings, one can pick up from the data division of a COBOL program some global invariants. In general, invariants are useful items

*Programming Research Group, Oxford University Computer Laboratory, 11 Keble Road, Oxford OX1 3QD, U.K.
†Performance Technology Group, Lloyds Register of Shipping, Lloyds Register House, 29 Wellesley Road, Croydon CR0 2AJ, U.K.
REengineering, DOcumentation and Validation of systems, ESPRIT project no. 2487.
of high-level information, being statements of the relationships between run-time values, such as \( \text{sum} \geq 0 \) or \( 0 < n \leq \text{acct}_{\text{tot}} \Rightarrow \text{acct}_{\text{rec}}(n) \in \text{acct}_{\text{file}} \) (literally ‘if \( n \) is in the range \((0, \text{acct}_{\text{tot}}]\), then the \( n \)'th \( \text{acct}_{\text{rec}} \) is listed in the \( \text{acct}_{\text{file}} \)'). For example consider a ‘level 88’ condition name:

\[
\begin{align*}
02 & \ V \ PIC \ X. \\
88 & \ A1 \ VALUE \ IS \ B1. \\
88 & \ A2 \ VALUE \ IS \ B2.
\end{align*}
\]

where \( B1 \) and \( B2 \) are literal values in the type \( X \) (which denotes the type of single characters in COBOL). One can read this as asserting that the properties

\[
(V = B1) \iff A1 \\
(V = B2) \iff A2
\]

hold invariably throughout the program, i.e. that \( A1 \) is a boolean which may be treated as an abbreviation for the condition-test ‘\( V = B1 \)’, and \( A2 \) likewise.

One also gets invariants (although more trivial ones) from ‘redefinitions’ (or ‘casted equivalences’, more properly). For example, the declaration

\[
\begin{align*}
\text{LevNum} & \ A \ PIC \ F1. \\
\text{LevNum} & \ B \ PIC \ F2 \ REDEFINES \ A.
\end{align*}
\]

asserts that, throughout the program

\[
b = [F1 \rightarrow F2](a)
\]

holds true, where \( a, b \) are the values returned by a use of \( A, B \) respectively, and \( [F1 \rightarrow F2] \) is the function performing translations of quantities with format \( F1 \) to quantities with format \( F2 \). These examples illustrate the kind of information that automatic tools can pick up from the raw COBOL source of an application program on their own, but it is not enough. One wants to produce meaningful abstractions from the source, and to this end we have developed a method which is summarised in Figure. Software tools are used to progressively transform the application code into object-oriented abstractions.

The three stages of the process (‘clean’, ‘specify’ and ‘simplify’ in the terminology of [2]) are:

Stage 1: (‘clean’) Translation to the intermediate language UNIFORM, eliminating redundant language constructs. For example, ‘\( \text{MOVE} \ X \ \text{TO} \ Y \)’ becomes ‘\( Y := [FX \rightarrow FY](X) \)’, where \( FX \) and \( FY \) are the types of \( X \) and \( Y \) respectively. Also, asserted relationships between data values (as in the ‘REDEFINES’ statement above) are translated into statements about invariants in the programs run-time behaviour. This stage can be seen as restricting the original language to a small subset of permissible constructs, because UNIFORM is not an inherently different language to COBOL, just more compact.

Stage 2: (‘specify’) Using data-flow diagrams for guidance, associated variables are grouped together to create prototype objects – these are object-oriented entity descriptions consisting of lists of attributes and their types, perhaps with initial values, but as yet containing no list of associated operators.

The code is also split into phases at this point – sections within which no files are opened or closed, or have their read/write status changed. Equational descriptions of the functionality
Stage 3
Implementation

Stage 2
Internal Design

Stage 1
External Design

Requirements

Object-Oriented Design

Detailed Func. & Prototypes

Low-level Design Intent

Modular plan of Data, Func.

Information

Language

Source Code

COBOL

Z++

UNIFORM IL

Detailed Functional IL & OO Decl.

Reverse

Forward
associated with these phases are obtained automatically and transcribed into the intermediate functional language, simplifying transforms being automatically applied in order to reduce the equational presentation to a normal form\(^2\).

Stage 3: (‘simplify’) The abstracted functional descriptions are incorporated into the outline objects as descriptions of their operations, thus filling in the semantics of the prototype objects identified at Stage 1. A full specification (in the language Z [13] or Z\(^++\) [6]) is then printed out using the object-orientated abstraction as a basis, along with associated textual documentation.

and the plan of the rest of this article is as follows. We first of all set out the ‘three stages on the path to abstraction’ in more detail in the next section, then collect together some ancilliary points in a final section.

## Contents

1 The Three Stages along the Path to Abstraction 4

  1.1 Stage 1: Translation to UNIFORM ................................. 5

  1.2 Stage 2: Finding Higher Abstractions .......................... 8

      1.2.1 Collecting Together Data and Functions .................. 8

      1.2.2 Phases .................................................. 9

      1.2.3 Slices .................................................. 9

      1.2.4 Heuristics for Code Purpose Determination .............. 10

      1.2.5 Business Rules and Implementation Operations .......... 10

      1.2.6 Functional Abstraction and Transformation .............. 11

      1.2.7 Examples .............................................. 12

      1.2.8 Modifications to Objects ............................... 12

  1.3 Stage 3: Final Steps – Z and structured documentation .... 15

      1.3.1 Production of Summaries ............................... 16

2 Further Points 17

  2.1 Other Languages .............................................. 17

  2.2 Change Requests ............................................. 17

  2.3 Presentation of Equations as COBOL ......................... 17

  2.4 Comparison with other techniques ............................ 21

1 The Three Stages along the Path to Abstraction

In the following sections we describe the three stages outlined in the Introduction in detail. But first a word on where we are going: we want to turn an application code into an object-oriented program, in which the definitions within each class (generic object type) are couched in a high-level specification language (here, Z [13]). The application code will then reduce to a series of calls for an object method to be applied to an object. It will be much shorter, the bulk of the ‘program’ having shifted to lie within the class definitions. If one chooses to regard the classes as opaque, then one will have achieved an abstraction away from the details of the implementation, and if one does not wish to ‘forget’ the definitions within the classes, one can go on to provide

---

\(^2\)see Section ?? in this book.
real abstractions of each operations semantics, simply by replacing the captured functionality with a looser specification.

For a full description of a COBOL application however, we have to construct classes of objects for the standard COBOL concepts such as files and sequences. This is a necessary preliminary – one could look up the descriptions in a manual if any such existed, but failing such a source, we give some examples here, as an indication of the kind of description we would expect to obtain from the rest of the application code too. In fact, COBOL provides seven varieties of file:

- **sequential**
- **relative**, with either
  - sequential,
  - random or
  - dynamic access, and
- **indexed**, also with
  - the above three modes of access (access mode cannot be changed within a program).

and each of these types of files is describable as an **object class**. Each class interprets the operations RECEIVE, SEND, etc., in its own particular way (the advantage of an object-oriented model is that the meaning of a ‘command verb’ in the code will be dependent on the object to which it is applied, but the semantics will nevertheless be unambiguous and, it is to be hoped, will correspond to the natural meaning of the name chosen for the operation).

The description of dynamic files is shown in figure 1 (when ALTERNATE KEY’s are to be allowed for the file [12], an extra initial operation ‘SETKEY’ which sets a given key to be the access key will be included amongst the operations of the class). The description language is Z++ [6] – an object-oriented enhancement of the specification language Z [13]. It will be seen that the specification language allows one to use abstractions such as sets – the ‘{...}’ notation – and functions, and provides names for relations like ‘x ∈ y’ (x is a member of set y) and operators like ‘a ◦ f’ (restriction of function f by removing the part a of its domain set) and ‘f ⊕ g’ (overwrite function f with function g on the points in the domain set where g is defined).

For each class of file described, one has to settle on a suitable carrier type for the internal data structure, and then the operations are described as changes to the contents of the structure. We use **functions** to represent the internal structure of indexed files and sequences to represent sequential and relative files, and these basic representations are to be preferred over some uniquely tailored structure because this reduces the number and complexity of possible operations that need to be considered overall (and whose properties need to be known during validation).

The COBOL REPORT WRITER can also be described as a class in much the same way. The class takes as parameters list of associated record types, and contains the operations INITIATE, GENERATE and TERMINATE.

### 1.1 Stage 1: Translation to UNIFORM

Our reverse-engineering toolset first of all translates COBOL code into UNIFORM [14] intermediate code. In the process (which is automatic on loading a COBOL application) it generates and stores additional information, such as the initial values of variables. The resulting UNIFORM code is semantically equivalent to the original, and forms the basis for all further reasoning.
class reldynfile|Data|
  types */ locally defined types */
    Status def = {closed, input, output, i_o}
  owns */ local variables */
    contents : sequence|Data|;
    fkey : N ; /* the natural numbers */
    frec : Data;
    status : Status
invariant
  fkey ∈ dom contents ⇒ frec = contents( fkey)
returns
  AT_END : → Bool ;
operations
  OPEN : Status → () ;
  DELETE : () → () ;
  READ : () → Data ;
  WRITE : Data → () ;
  REWRITE : Data → () ;
  CLOSE : () → () ;
actions */ operation semantics */
  OPEN x def = x = i_o ⇒
    fkey' = 1
    status' = i_o
    x = input ⇒
    fkey' = 1
    status' = input
    x = output ⇒
    fkey' = 1
    contents' =
    status' = output
    /* x = closed ⇒ False */
  AT_END def = fkey > # contents;
  WRITE x def = status = output ⇒
    contents' = contents ⊕ {fkey ↦ x};
    /* status ≠ output ⇒ False */
  READ x def = status ∈ {i_o, input} ⇒
    x = contents(fkey) ;
    /* status ∉ {i_o, input} ⇒ False */
  DELETE def = status = i_o ⇒
    contents' = squash( {fkey} ∪ contents);*
    /* status ≠ i_o ⇒ False */
  SET n def = fkey' = n;
  REWRITE x def = status = i_o ⇒
    WRITE x;
    /* status ≠ i_o ⇒ False */
  CLOSE def = status' = close
end class

Figure 1: The object class reldynfile corresponding to COBOL ‘relative dynamic’ files.
One can carry out what the tool does quite easily oneself. The intention is simply to eliminate the redundant constructions in COBOL and replace them with a smaller and clearer set.

- Re-express all `PERFORM UNTIL` statements as `DO UNTIL` loops. `PERFORM TIMES` and `PERFORM THRU` statements can also be eliminated, at the cost of introducing a new local variable.

- Eliminate the `DEPENDING ON` clause in `GO TO` statements by using an explicit `IF ... THEN` construct.

- Transform all arithmetic verbs (ADD, SUBTRACT, COMPUTE, and so forth) into assignments – or conditional assignments, when it is necessary to take account of `ON SIZE ERROR` clauses.

- A basic complication of COBOL is the significance of data formats in `MOVE` and other statements. We have to re-express a statement

  \[
  \text{MOVE } X \text{ TO } Y
  \]

  where X and Y are variables, with formats FX, FY respectively, as the explicit assignment

  \[
  Y := [FX \rightarrow FY](X)
  \]

  and not as just ‘Y := X’. Here, for example, \([X[3] \rightarrow X[2]](\_)(X[n] \text{ is the type of length } n \text{ character arrays})\) denotes the function which left-aligns the input characters, possibly losing the rightmost character.

- A function `no_overflow_F(\_)` must also be used to express the fact that COBOL looks to see if an overflow error occurs when a specific data item is moved into a variable with format F. It is often necessary to introduce this in order to express arithmetic calculations in COBOL.

- Replace `SORT` statements by calls of a stream-based function of the same name, as shown below. The function receives an input \(P_{1\text{out}}\) (the stream of records released by the procedure or file P1), sorts it according to the sort specification, and returns it in sorted order as \(P_{2\text{in}}\) (the stream of records to be read by the procedure or file P2). If the COBOL statement specifies that the work will be done in some specified temporary file space (sequence of records), say `tempfile`, one identifies the initial sequence of records in `tempfile` with the output from `P1` before the sort, and its final state with the input to `P2` after the sort. Thus a statement

  \[
  \text{SORT } \text{tempfile} \\
  \text{ON } (\text{sort_spec}) \\
  \text{INPUT } \text{PROCEDURE IS P1} \\
  \text{OUTPUT } \text{PROCEDURE IS P2}.
  \]

  will be abstracted to

  \[
  P_2(y', P_{2\text{in}}) \\
  \text{ where } \\
  P_{2\text{in}} = \text{SORT}(P_{1\text{out}}, \text{sort_spec}) \\
  (y', P_{1\text{out}}) = P_1(y) \\
  \text{tempfile} = P_{1\text{out}} \\
  \text{tempfile}' = P_{2\text{in}}
  \]
(the \( y \) are the program global variables, written out explicitly). The information here is that the \texttt{SORT} has no side-effect on the program variables, but depends on the \texttt{sort_spec} for its precise action. In contrast, \texttt{P1} and \texttt{P2} can have side-effects on the program globals.

The procedures \texttt{P1} and \texttt{P2} will contain COBOL \texttt{RELEASE} \langle \texttt{rec} \rangle \texttt{TO workfile} statements and \texttt{RETURN} \langle \texttt{rec} \rangle \texttt{FROM workfile} statements respectively. If files \texttt{F1} and \texttt{F2}, rather than procedures \texttt{P1} and \texttt{P2}, are used as the inputs or outputs to a \texttt{SORT}, we simply replace \texttt{P1} with \texttt{F1} and \texttt{P2} in with \texttt{F1}.

- A similar replacement takes place for \texttt{MERGE}, a command which takes a set of files (not input procedures) and produces a new merged file.

- Replace calls to an operating system, or to a transaction processing system such as CICS, by calls to object-oriented operations. The environment systems can each be represented as (single object) classes, with private variables which can only be changed from the program by means of calls of the operations of the class (an outline description of the CICS API system in this style is given in [9], and provides a concise way to define the necessary parts of these environments).

Note that any ‘callable subprogram’ in the COBOL source also corresponds to an object. The class in this case has two operations: \texttt{CALL} and \texttt{CANCEL} (the latter to reset the values of the sub-program’s variables to their initial values, corresponding to a COBOL command \texttt{CANCEL (subprog)}). Different instances correspond to different subprograms, and accordingly, have different codes which are executed when a \texttt{CALL} call is made. Of course one cannot set up these objects before examining a particular application code.

Overall, the transformation into UNIFORM can be viewed as a translation into a small subset of COBOL. The aim is simply to facilitate the later stages of abstraction by reducing the number of constructs that those stages need to deal with, and by making more explicit certain details which are only implicit in the COBOL text. This will reduce the need to later correlate information from different sources, for example, the \texttt{data division} and the \texttt{procedure division}.

### 1.2 Stage 2: Finding Higher Abstractions

Examination of the \texttt{file section} and \texttt{environment division} yields the files of the program and their associated flags and counters, but the precise connection and arrangement has to be established by looking at the UNIFORM intermediate language translation. Data-flow diagrams give information on what might be the outline shapes of objects in the application code: we first of all designate the files, indexed arrays and reports as \textit{main variables} of an application – they will be represented as objects with themselves as the chief attribute – and then try to pick up the flags, counters and so on which are associated with them. The details of the procedure are set out below.

#### 1.2.1 Collecting Together Data and Functions

Looking at the data-flow analysis of the UNIFORM translation of the program allows us to find the variables logically associated with the main data structures of the program, as well as the data-flows between the main variables themselves. For instance, if there is data-flow from a simple variable \( X \) into a file \( F \)

\[
X \rightarrow F
\]
and \( F \) is the only variable which receives an immediate data-flow from \( X \), then we must produce an object corresponding to \( F \) that has

- \( X \) as a hidden attribute \( F.X \), and
- an operation \( \text{foo} \) that takes an input value of the same type as \( X \), and updates the object:

\[
F.\text{foo}(x) \overset{\text{def}}{=} F.X := x
\]

- similarly, for output, an operation \( \text{bar} \) which reports the value of \( X \) to the outside world:

\[
F.\text{bar} \overset{\text{def}}{=} \text{return}(F.X)
\]

A global data flow from \( F \) to another main variable \( G \)

\[
F \rightarrow G
\]

suggests a high-level function \( \text{foo}_{F,G} \) – a method \( \text{foo} \) of \( G \) which uses some function \( \text{bar} \) of \( F \) to update \( G \) internally:

\[
G.\text{foo} \overset{\text{def}}{=} \ldots F.\text{bar} \ldots
\]

where the description of \( \text{foo}_{F,G} \) as dependent on \( F \) and \( G \) both arises because the action of \( \text{foo} \) depends on the object \( G \) to which it is applied. In this object one finds a description of the action which refers to the class of \( F \).

In this way one gradually adds new attributes to the basic objects that describe the types of the main variables as one goes through an application program. One also has to add in invariants which (it is believed) link these attributes to the existing attributes of the file. For example, with a suitable definition of the predicate \( \text{atend} \), the statement

\[
\text{csfatend} = \text{atend}(\text{accountfile})
\]

asserts that the variable \( \text{csfatend} \) serves as the ‘end of file flag’ for the file \( \text{accountfile} \). Whenever the file pointer points to the end of the file, the value in \( \text{csfatend} \) will be ‘\text{True}’. These ‘believed invariants’ will eventually need to be checked and validated.

A sensible order of working using the abstraction tools is as follows:

- for each putative object class, identify global functions (represented by data flows between the main structures of the program) whose destination is the main structure in the class.

- Each such data flow identifies a candidate operation that updates the present class, and one then undertakes a more detailed analysis in order to find the precise functionality and the meaning of this operation.

and other techniques which aid in breaking up the functionality of the program into manageable chunks are detailed below.
1.2.2 Phases

One should always attempt to split the program into phases before beginning a semantic analysis. A phase is a maximal logically connected piece of code which contains no OPEN or CLOSE statements.

A phase represents an execution period in which the files are processed in a particular mode (append, overwrite, etc.), and hence should represent a single input-output function on these files. The modalities of the files during a phase give a clue to what the overall form of this function could be (see Section 1.2.4), and the details can then be extracted from the code of the phase by more detailed functional abstraction.

1.2.3 Slices

If useful phases cannot be identified (if, for instance, all OPEN statements are in an initiation paragraph, all CLOSE statements in a termination paragraph), one should use program slicing on the variables of an object \( C \) — and those variables already identified as likely to be used in a method of \( C \) — to distinguish the code which touches these variables from that with irrelevant functionality.

For a set of variables \( G \) and code \( C \), the operation \( \text{slice}_G(C) \) returns a piece of code that on the variables \( G \), has the same functionality as \( C \). For instance,

\[
\text{slice}_{\{f\}} \left( \begin{array}{l}
x := y + 10 \\
z := x * x \\
\text{SEND } x \text{ TO } f
\end{array} \right) = \begin{array}{l}
x := y + 10 \\
\text{SEND } x \text{ TO } f
\end{array}
\]

Once this code has been obtained, one uses (the tool for) functional abstraction to simplify the description of the function corresponding to the slice.

1.2.4 Heuristics for Code Purpose Determination

In order to get an idea of the basic intention of a program or sub-program, we categorise according to its modalities, as determined during the analysis into phases. Table 2 summarises the plausible purposes of a piece of code, working from the number of input and output files and streams.

<table>
<thead>
<tr>
<th># Inputs</th>
<th># Outputs</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Arithmetic function, or string manipulation</td>
</tr>
<tr>
<td>1 (stdin)</td>
<td>0</td>
<td>Building up data in an internal table or value</td>
</tr>
<tr>
<td>0</td>
<td>1 (stdout)</td>
<td>Output routine</td>
</tr>
<tr>
<td>1 (file)</td>
<td>1</td>
<td>Data extraction, summarization, or rearrangement</td>
</tr>
<tr>
<td>1 (stdin)</td>
<td>1</td>
<td>Building up data in file</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Data validation or data extraction</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Master-transaction updating</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Master-transaction updating with error reports</td>
</tr>
</tbody>
</table>

Figure 2: The likely purpose of code according to overall form

These are only the most plausible explanations of the purpose, further details are needed to decide whether these are reasonable descriptions or not. For example, are there flags which select
whether records are to be inserted, deleted or updated in the old file, do the files have suggestive names – like OLD-MASTER, NEW-MASTER, etc.? Clearly there is scope here for the application of knowledge based techniques and more sophisticated heuristics. At least COBOL does have certain high-level functions built-in, such as SORT, MERGE and SEARCH, so simplifying some of this analysis.

The output from this stage of the abstraction process is a set of comments attached to the named slices or phases.

1.2.5 Business Rules and Implementation Operations

There are two different design aspects in most programs: there are

- *embedded business rules*, such as ‘derive the maximum allowable loan for account x’, which operate at a low level, typically on a per record basis. These are at bottom of the
- *implemented operations*, such as a file-processing procedure, which typically incorporate several different business rules in the operations on each record of the file.

and our approach enables us to isolate both these components of the functionality.

The implementation operations are identified by enumerating the phases and data-flows of the program, and then are broken down into logically coherent suboperations by inspection of the derived functional descriptions. Some of the suboperations will correspond to business rules, usually of the form, ‘if X then Y = Z’. A description of the procedure which abstracts the functionality of a source program follows below.

1.2.6 Functional Abstraction and Transformation

We assume now that we have identified suitable sections of translated code for further analysis. This code is (‘dirty’) UNIFORM, that is, it still contains the PARAGRAPH’s and GO TO’s of COBOL, and it retains a fall-through semantics which complicates reasoning about it. One converts this into a set of equations which give the mathematical meaning of the code, set out in a simple ‘functional intermediate language’. The process is described in detail in Section ???. Here we note only that there are just two essential constructs: functional application, ‘f(a_1,\ldots, a_n)’, and conditional expressions, ‘cond(e, a, b)’, also written as ‘e \rightarrow a, b’, which return value a if e is true, b otherwise.

The reverse engineer has some choice over how paragraphs and loops and sections in the original program are represented as functions: one may either

- produce a new function L for every loop in the original program (which is defined then by a tail recursion in the abstraction), and a function P for every paragraph P, which expresses the ‘fall through’ semantics of P. A GOTO P will then be represented simply by a call of P, but a PERFORM P will be more complicated to represent. Or one may choose to

- represent the PERFORM semantics of a paragraph or section P as the named function P. This differs from the fall-through semantics in that it does not go on to execute the rest of the program following it, but instead RETURNs at the end of paragraph. Then a PERFORM P call is simply represented.

In some cases it will be necessary to use the PERFORM semantics in order to avoid an infinite series of expansions as the abstraction transformation replaces PERFORM statements by the code of
the performed paragraph. Such infinite expansions will occur if there is a loop in the paragraph call graph and if all of the calls in the loop are PERFORM calls rather than GO TO calls.

Having obtained the equations which describe the program, one then simplifies them. A set of normalising transformations are described in [1], and these are applied after every high-level modification to produce a consistent and standard form for the user. Afterwards, one may apply the following higher-level transformations in order to simplify the equations:

1. Substitution of a function body for its calls (‘unfolding’ in the terminology of [4]).
2. Eliminating recursion by recognising a simple iteration.
3. Adding in new equations, either as properties to be proved, or as assumptions to simplify the description.
4. Using invariants or other equations to substitute one expression for another inside a function description (‘folding’).
5. Renaming a function.
6. Deleting a function.
7. Proving properties of a function (including induction when the function is recursively defined).

See Section ?? for a fuller description.

These operations have been built into the reverse engineering tool produced at Oxford. In (3) new equations are added to the list of global hypotheses (assertions still to be proved) which pertain, whereas in (7) the property will be removed from the global hypotheses after successful proof.

Validation of the equations describing the meaning of the code against some set of specifications can be carried out mechanically using the reverse engineering tool by adding properties of the form $P(f(v))$ to the set of equations (where $f$ is one of the functions defined by the equations), and using the transformations listed above (this is equivalent to more conventional methods of generating preconditions, although it does not directly provide documentation.)

### 1.2.7 Examples

As an example, the COBOL program in figure 3(a) is not immediately comprehensible for a variety of reasons (below). but after translation to UNIFORM it becomes as shown in figure 3(b). It has the functional abstraction in figure 3(c), and after substitution, the definition of $p_1(X)$ simplifies to:

$$p_1(X) = 10$$

and note that the equation $p_1(X) = p_2(5)$ in the abstraction is not a mistake, even though the code for paragraph $p_1$ begins with ‘MOVE 0 TO X’, which might lead one to suppose that it then executes paragraph $p_2$ with $X$ set to 0, not 5! In fact, the final line in the code for paragraph $p_1$ specifies a preliminary operation ‘PERFORM p2 THRU p4’ prior to executing $p_2$, and it is this preliminary which sets $X$ to 5. It is merely coincidental that the first action of the preliminary is to ‘PERFORM p2’ also.
Figure 3: (a) COBOL program. (b) UNIFORM translation. (c) Functional abstraction.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT extf ASSIGN TO file
ORGANIZATION IS RELATIVE
ACCESS MODE IS DYNAMIC
RELATIVE KEY IS ptr
FILE STATUS IS status.

DATA DIVISION.
FILE SECTION.
FD file LABEL RECORDS OMITTED.
 01 frec.
    02 height PIC 9(3).
    02 weight PIC 9(3).
WORKING-STORAGE SECTION.
  01 ptr PIC 9(5).

PROCEDURE DIVISION.
a SECTION.
b.
OPEN I-O file.
READ file AT END MOVE true TO eof.
PERFORM filter UNTIL eof.
CLOSE file.
STOP RUN.
filter.
IF (weight > 250) DELETE file.
ADD 1 TO ptr.
READ file AT END MOVE true TO eof.

Figure 4: (a) An example COBOL program which involves file-handling...

INITIALLY : eof = false
INVARIANT: frec = string(height, weight)
            file = extf

EQUATIONS:
  b(file, frec, ptr, eof) = (file, frec, 1, true) if atend(file)
                             = loop(file, file(1), 1, eof) otherwise.

  loop(file, frec, ptr, eof) = (file, frec, ptr, eof)
                               if eof
                               = loop(DELETE file(ptr), frec, ptr+1, true)
                                   if weight > 250 ∧ atend(DELETE file(ptr))
                                   = loop(DELETE file(ptr), DELETE file(ptr)(ptr+1), ptr+1, eof)
                                   if weight > 250
                                   = loop(file, frec, ptr+1, true)
                                   if atend(file)
                                   = loop(file, file(ptr+1), ptr+1, eof)
                                 otherwise.

Figure 4: (b) ...and the initial abstraction of functionality.
An example involving files is the program shown in figure 4(a). This code is abstracted to
the description in figure 4(b), from which we prove the invariant:

\[ \text{eof} = \text{atend(file)} \Rightarrow \text{eof}' = \text{atend(file')} \]

where \( \text{eof}' \), \( \text{file}' \) are the actual values of these parameters in the nested calls of \text{loop}.

```
class file_class
    extends reldynfile[type1][ptr/fkey]

owns
    eof : X[1];
    weight: N [3]

invariant
    eof = atend(contents)
    frec = string(height,weight)

operations
    ELIMOW: () \rightarrow ()

actions
    ELIMOW def
    (contents',frec',ptr',eof') = loop(contents,contents(ptr),ptr,eof) if \neg\text{eof}
    = (contents,frec,ptr,true) otherwise.

end class
```

Figure 5: The Object Class abstracted from the COBOL source.

There is only one phase, with mode:

\[ \text{file} \rightarrow \text{file} \]

so we derive the outline class in figure 5. The program itself becomes, after replacement of the
scattered parts of the object-oriented functionality with explicit calls:

```
type1 : ....;
file : file_class
begin
    OPEN I-O file;
    ELIMOW of file;
    CLOSE file
end
```

and all the operational details have moved to the interior of the definition of the object class.

1.2.8 Modifications to Objects

The example above demonstrated the modification of a program by replacing its scattered
variables by single objects and attributes and calls on objects. We can then transform these objects,
using the refinement rules derived in [7] to simplify their presentation and eliminate redundan-
cies. The class hierarchy structure gives information on the design of a system, and will provide
suggestions for improvement and modification of this design, for example, by making two similar
methods instantiations of a parameterised and more general method.
1.3 Stage 3: Final Steps – Z and structured documentation

From the set of functions that describe the functionality of a code section, we can derive a Z specification, whose state is the set of declarations used in the code, constrained by the known invariants, and whose operations are the individual functions of the code.

Each of these Z descriptions can then replace the function inside the object class within which it is defined, so that we obtain a Z++ description of the overall program. This is a set of classes, whose internal actions are described by means of (a subset of) Z.

A lot of detail from the original program has been lost in this description – for instance, which order certain assignments occurred in, or whether GO TOs or PERFORMs were used to implement a recursion, and so forth. But knowledge has been accumulated and integrated, so that the functions and operations of the code are now clearer and closer to the level at which a person unfamiliar with the source code could understand and modify. Reuse is also enhanced, as the object operations have been separated from their particular use in the given program (which is now just a series of object method calls).

Further documentation can now be produced: from the objects we can extract whole/part graphs [5], which reveal the class inheritances, and data flow graphs for each method, corresponding to SSADM data flow graphs. Entity life histories can be produced from the transformed source code program to show the order in which particular operations are performed. Thus we can present the extracted design of the application in a way that requires no mathematical notation, with simply a graphical presentation of object hierarchies and method input-output modalities.

Metrics are available from the data flow graphs produced in the first phase of abstraction: these give an indication of the complexity of variable interactions in the code: the weight of each variable (the number of vertices in the data flow graph that are incident with it) gives an indication of how difficult it may be to change that variable or its roles, or to comprehend these roles. Thus both the average weighting of variables throughout the program, and abnormally high weightings should be examined. A weighting of 0 of course means that a variable is redundant (logically at least).

We can also give suggestions for improving the positioning of file OPEN and CLOSE statements, for instance, moving them so that files are open for the minimum extent of program scope necessary: which is useful for code comprehension in addition to being safer. We would expect that only one input-output function is represented in each phase, that is, the subset of the data flow graph of the phase determined by the active file variables, is connected.

As an example of the production of Z from code, we take the factorial function:

```plaintext
i: integer;
x: integer;
begin
  x:=1;
  do while (i > 1):
    x := x*i;
    i := i - 1
  end do
end
```

which becomes abstracted to:
which can be pretty-printed as the set of $\mathbb{Z}$ declarations:

\[
\text{STATE} \\
\text{x : } \mathbb{R} \\
\text{i : } \mathbb{R}
\]

\[
\text{loop : } \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R} \\
\forall x : \mathbb{R}; i : \mathbb{R} \bullet \text{loop}(x, i) = \text{loop}(x \times i, i - 1) \text{ if } i > 1 \\
= (x, i) \text{ otherwise.}
\]

\[
\triangle \text{STATE} \\
(x', i') = \text{loop}(i, i - 1) \text{ if } i > 1 \\
= (1, i) \text{ otherwise.}
\]

1.3.1 Production of Summaries

Each program will give rise to a summary of the form:

The main structures of the program are:

... 

The functions on these structures are:

... details of inputs and outputs for each identified function ... 

The business rules or purpose associated with these functions are:

... details of the rules and identified purpose of each function ...

2 Further Points

In this section we collect together some points which are valuable in themselves, but which do not form part of the ‘how to’ guide which we have tried to set out in the previous section.

2.1 Other Languages

Although the techniques have been devised with COBOL, and specifically COBOL 74, as the priority, the processes of object recognition and functional abstraction apply to any procedural language. Techniques originally designed for UNIFORM have been adapted to deal with COBOL, however a generic sublanguage of both COBOL and FORTRAN would be a useful intermediate step in the production of abstractions for these languages, and UNIFORM can serve this purpose in our system. A large subset of PASCAL could also be handled by our approach.
2.2 Change Requests

We can give guidelines for using the reverse-engineered code to facilitate modifications requested by users or by internal enhancement decisions. Each identified high-level function will possess an attached comment, stating what it does, and what business rules it is associated with. A modification request will lead to a specific functional modification request upon one or more of these rules, and hence to the functions that implement them. An enhancement request, for additional functionality, will require the identification of a suitable set of input and output structures, and this will provide a list of classes in which the new functionality can be expressed. Environmental modifications are interpreted as modifications to the low-level classes that represent the operating system or specific devices. By representing these interactions at a sufficiently abstract level, we have reduced the dependence of the application on a particular operating environment, and we will require only certain abstract algebraic properties of these environment classes to be preserved under the modification for the correctness of the design to be maintained.

2.3 Presentation of Equations as COBOL

It is possible, to an extent, to present the derived abstractions as COBOL or psuedo-COBOL code. For example, a recursive function of the form

\[ f(x) = a(x) \quad \text{if } e \\
= f(b(x)) \quad \text{otherwise.} \]

can be expressed (with more prolixity than is strictly necessary, in order to show the generic technique) as

\[
\begin{align*}
  & f. \\
  & \text{PERFORM } f0 \text{ UNTIL } e. \\
  & \text{PERFORM } f1. \\
  & f1. \\
  & \text{COMPUTE } x = a(x). \\
  & f0. \\
  & \text{COMPUTE } x = b(x). 
\end{align*}
\]

in COBOL, where \( e \) is the appropriate COBOL expression.

In general, the technique for producing pseudo-code relies on the normalization procedure [1] to produce equations of a very specific shape. Because, after normalization, the equations are not as arbitrary as they might be, the technique for converting them can be made quite simple in design. An equation in normalised form [1] will have a rhs which is either

1. a nontrivial conditional \( b \rightarrow e_1, e_2 \), for some further (vector) expressions \( e_1, e_2 \) and a simple (scalar) boolean expression \( b \), or

2. a simple vector expression \( e \) which does not contain any conditionals. These take the form \( (v_1, \ldots, v_n) \) (possibly with \( n = 1 \)), where the \( v_i \) are each scalar expressions without conditionals, which are either

   \[ i\] the application of an atomic scalar function \( \text{foo} \) to its arguments: \( \text{foo}(a_1, \ldots, a_n) \), where the \( a_i \) are further simple 1-dimensional expressions without conditionals, or
[ii] an infix operator expression \( e_1 \ op \ e_2 \) between two simple scalar expressions without conditionals, or

[iii] an atomic scalar constant or local parameter, e.g. \( \pi \) or \( x \), or a reference to the scalar components of atomic vector values, e.g. \( b[2] \).

and we can easily say how to translate an equation of this form into COBOL. Note that the \( \text{lhs} \) of an equation is either

(a) a name without parameters, e.g. ‘a’, defining a (possibly vecror-valued) data item, or

(b) a function name followed by one or more variable names standing in for its parameters: \( f(x_1, \ldots, x_n) \), defining a 1-dimensional function with arguments. \( \text{N-dimensional functions} \) can be regarded as \( \text{n 1-dimensional functions} \).

and the two cases have to be treated differently. Below we sketch the technique for producing the pseudo-COBOL translation.

We deal with the case (a) of an equation defining a data item (a non-function) first. There is then no possibility of encountering a local parameter name in the case (2iii) for a \( \text{rhs} \).

To translate an equation ‘\( a = \ldots \)’, where \( a \) is a vector quantity, first of all we allocate storage within the data division:

\[
01 a. \\
02 a1 PIC 9(16). \\
\ldots \\
02 an PIC 9(16).
\]

for example, assuming that the vector components each require 16 bytes.

Next, we write a paragraph in the procedure division which fills it with the correct value. To save time in explanations here we call the paragraph by the same name, ‘a’. If the defining equation was a conditional \( b \rightarrow e_1, e_2 \) (case 1), the paragraph contains code which calls two unique subparagraphs \( a1 \) and \( a2 \) corresponding to \( e_1, e_2 \), depending on the result \( b \) deposited in a fixed global variable (‘EXPRVAL’ say) by a paragraph \( a0 \). So ‘a’ looks like:

\[
a. \\
\text{PERFORM a0.} \\
\text{IF EXPRVAL NOT = 0} \\
\text{PERFORM a1} \\
\text{ELSE PERFORM a2.}
\]

We define the paragraphs \( a0, a1, a2 \) according to the structure of the expressions \( b, e_1, e_2 \) too – but \( a0 \) at least will not have any conditionals in it, although \( a1 \) and \( a2 \) might.

Eventually the naming convention descends to using names for paragraphs like ‘a1212112’ and no more conditional tests occur within paragraphs (case 2). The paragraph simply corresponds to an unconditional vector expression \( (v_1, \ldots, v_n) \), and contains code which calls subparagraphs which calculate the scalar components of the vector in turn and put them in the correct place in array \( a \). Each of these stores its scalar result in ‘EXPRVAL’, and the paragraph ‘a1212112’ which fills the vector looks like this:

\[
a1212112. \\
\text{PERFORM a12121121} \\
\text{MOVE EXPRVAL TO a1}
\]
Now all one has to do is translate simple scalar expressions. Function applications (case 2i) are translated to paragraphs which first of all fill a vector with the argument values, then apply the function. This requires a certain amount of storage space to be set aside for 'concurrent' calculations for the argument values, which cannot all take place in the single storage variable 'EXPRVAL'. If the function foo takes n arguments, then one has to declare:

```
01 fooPARAMS.
  02 fooPARAM1 PIC 9(16).

... 02 fooPARAMn PIC 9(16).
```

in the data division. One then can translate function applications of built-in's, such as sin (the sine function), for example:

```
a12121123.
  PERFORM a121211231
  MOVE EXPRVAL TO fooPARAM1

...  PERFORM a12121123n
  MOVE EXPRVAL TO fooPARAMn
  COMPUTE EXPRVAL = foo(fooPARAM1,...,fooPARAMn).
```

If, on the other hand, the function foo is defined locally in an equation, then we replace the 'COMPUTE EXPRVAL =...' in the above with 'PERFORM foo', on the understanding that we will have defined a paragraph foo which takes values cached in the global parameter array fooPARAMS and calculates a result value in EXPRVAL. It is perhaps safest to follow the latter rule always, and write a special paragraph for each built-in function foo, which says, in effect 'calculate foo(fooPARAMS) and put the result in EXPRVAL'. Obviously, infix operations (case 2ii) are a special (2-parameter) case of functions.

The only case (case 2iii) left to consider is that of scalar constants as subexpressions. Clearly one just writes a paragraph which returns their value in EXPRVAL. If they are a 1-dimensional component of another global, say b[3], one just returns b3 into EXPRVAL instead of (wrongly) b. For example:

```
a121211231.
  MOVE b3 TO EXPRVAL.
```

Now consider the case when the equation defines a function a with parameters. This will be translated as a paragraph which looks in the aPARAMS array for its arguments, and returns a value in EXPRVAL. The only extra translation required is to look up a variable x which is encountered as an atomic subexpression in the correct component of the aPARAMS array. This case did not have to be considered when the equation defined a data-item alone, because there were no local parameters.

Notice that one has to construct a separate copy of the PARAMS array unique to each paragraph representing a functional call. This is because an expression foo(...bar(...)...) needs the values placed in fooPARAMS for foos benefit to be undisturbed by those placed in barPARAMS afterwards (but before foo stops needing them) for bars benefit. So one cannot use the same
storage space for both fooPARAMS and barPARAMS, unless there are no nested calls of one from the other.

Once code has been produced according to the above recipe, however, one still has to make a second pass in order to disambiguate some remaining parameter clashes. One has to make a separate copy of the paragraph representing a function call of foo for each occurrence of foo in an expression. Say these are foo1, foo2, etc., then one calls paragraph foo1 instead of foo when translating the first occurrence, and foo2 instead of foo when translating the second, and so on. This requires one also to make extra copies fooPARAMS1, fooPARAMS2, etc., of the fooPARAMS array.

There are two reasons behind this:

- expressions like foo(...)foo(...) may occur, and therefore different parameter tables are required for the two (intertwined) calls, and secondly
- a COBOL paragraph does not behave exactly like a subroutine on execution. Nested calls will not cause a return from each call to the place the paragraph was called (‘perform’ed) from – instead the first (outer) call will return to the later (inner) calls return address. This is because the return addresses are not stacked but overwritten.

To combat the latter effect, one has to make sure that each paragraph is called from only one place. So one makes a different copy of the paragraph for each call statement, to be safe, and this amounts to a different copy for each occurrence of the basic function call in the generating equation.

The code has to be cleaned up considerably after the initial translation! One can replace PERFORM ⟨para⟩ calls which only appear once with the body of the paragraph (if the call occurs twice, it is probably shorter to leave the paragraph alone), and make many of the storage arrays used occupy the same physical space: a single physical space can be made to correspond to a single maximal anti-chain in the graph. An anti-chain is a maximum set of function names (nodes in the graph) with no direct or indirect call relationship (directed path) between any two. All the functions in an anti-chain on the call-graph can utilize the same physical space to store their parameter values, because they never call each other. Obviously the minimum physical storage required is the minimal number of (maximal) anti-chains which will cover the directed graph completely. This is an invariant which can be extracted from the graph alone.

The complete forward engineering program can be seen to follow the same ‘clean, specify, then simply’ plan [2] as the reverse-engineering program;

- ‘clean’: normalize the specifications to the standard form given in [1].
- ‘specify’: translate the normalized set of functional equations into COBOL, as outlined above.
- ‘simplify’: increase the readability of the generated COBOL code by replacing calls to paragraphs with their bodies when the paragraphs do not correspond to any high-level concept.

In the regenerated code, each function is represented by a paragraph, and a function call is represented by a PERFORM of this paragraph. So no GO TO statements are necessary.

Thus a limited form of re-engineering is possible; by reverse engineering, then forward engineering again. We are also investigating the automatic rewriting of equational abstractions to produce WHILE-IF forms of code (in which there is no mutual or multiple recursion). Much research on such transformations exists in the functional programming world [4, 3].
2.4 Comparison with other techniques

As is argued in [17], using objects in the reverse-engineered abstractions reduces the effort needed to understand the interactions between data, and the technique set out here achieves this: the number of global variables is reduced, and the data is partitioned into distinct objects, where it cannot be affected by remote operations. Our approach also incorporates that of Linger [10], in producing explicit functional abstractions of the ‘business rules’ and file to file mappings present in the code (for large application codes, prior partitioning of the application into objects at the data level, and into object methods and processes at the procedural level, must be essential, and this is advocated here).

But the strategy laid out here is distinct from that of the REFORM project [16], at least. There assembly code is first translated into a ‘wide spectrum language’, which contains both Pascal-like and Z-like sublanguages, and code transformations are performed on the Pascal-like language in order to achieve simplifications (our strategy is to perform transformations on the higher level language, not the lower). There is a large library of transformations, and some guidance about the applicable transformations is given at each point, but selecting the right sequence of transformations remains difficult and exacting. We have avoided the need for a large library of transformations by working at the more abstract level, where there are fewer constructs, and the effects of transformations are clearer. Fewer transforms are needed, and automation is straightforward. So the REFORM approach relies on the reverse-engineer’s understanding of conventional procedural code, whereas the assumption here is that simple mathematical equations will be more accessible than the procedural code (COBOL). It may be true, however, that a presentation in a more accessible procedural pseudo-code might eventually be preferable, and we have set out here a scheme for converting equations back to COBOL which can be adapted – in fact, eased! – to other procedural languages. Certainly, the REFORM strategy, if applied to COBOL, would result in the reverse-engineer being presented with an unmanageable number of choices of transformation at each step!

Conclusion

We have described techniques by which COBOL code can be abstracted to produce explicit mathematical descriptions of functionality, and object classes representing the design of the application. The resulting representation is suitable for the application of theorem-proving techniques, and for transformations to simplify and clarify the design. It is related to the well-known process view of systems, since the global functions identified as methods of the object classes correspond exactly to such processes: the output routines, for instance, become methods that update the class built around the output stream.

The overall design of the tools that use these techniques is itself object-orientated, allowing components, such as abstracted functions, to be named, and used as attributes or method definitions at any suitable point in the descriptions. Thus the system is very flexible and easy to use, although it does require a familiarity with mathematical notation. About 40% of COBOL has been treated (76 keywords), but the basic concepts will be applicable to the remainder of the language.
References


