# The Software Migration Barbell

Andrew Malton
Department of Computer Science
University of Waterloo

**Abstract.** *Software source migration is the reengineering task of deploying existing software in a new environment, by significant modification of the source code. This paper presents a model, called the "Barbell Model", which accounts for the author's experiences in carrying out semi-automated source migration tasks. The Barbell Model consists of three phases: source normalization, carried out in the source domain alone; "blind" translation; and target optimization, carried out in the target domain alone. By restricting as much of the semantic work as possible to the source or target domains, the difficult problems arising "in the gap" between source and target can be avoided.*

**Introduction**. It is very frequently necessary when managing software to respond to changes in the available supporting environment by "porting" or migrating the software to a new platform. The need to migrate typically arises either because the current platform or environment is obsolete and poorly supported, or from the perception that a new platform or environment provides a better future for the software. Either the maintenance cost would go down, or the business value of the software would rise.

Maintenance tasks are classified into adaptive, corrective, perfective, and preventive maintenance [L]. Within this classification, software migration may be seen as an adaptive task: *adapting* existing software to a new environment. However, the need to migrate may be in order to *correct* errors arising essentially from underlying support software, to *improve* the available facilities by deploying software in another environment, or to *prevent* future maintenance costs by continuing development on a better platform.

Software source migration assumes the following goals:

- make efficient use of the target environment
- preserve functionality
- be maintainable source

The requirement to make efficient use of the target environment distinguishes migration tasks from adaptation by, for example, building emulators or virtual environments in which the adapted software can run more-or-less unchanged. (I use Windows software on my Mac this way, with the help of VPC [X].)

Preserving functionality is foremost among the code owner's requirements. It distinguishes migration tasks from design-reuse adaptation, look-and-feel simulation, and

The characteristic goal of *source migration* which distinguishes it from software migration in general, is that of finally producing maintainable source. Meeting this goal means that compiler technology which is directed solely at functionality (e.g. preprocessors, *XXX* to C compilers, and the like) cannot be viewed as source migration. Similarly, if large amounts of new software must be added to simulate on the taget features from the source, it is not source migration. Such techniques are useful but add to the maintenance overhead of the resulting system. The goal in source migration is to *replace* the existing source artefacts with new ones that are native and idiomatic in the best sense in the target environment.

**Related Work.** There is not a great deal of systematic work in this area.

There are a number of experience reports [F] [K] [G], explaining what was done in a particular project. For some of these [F] [G] the goal of maintainable source is not adopted: they are really compilers with high-level code generation. For source migration it seems that there is a wealth of *ad hoc* techniques, but not yet a systematic approach.

There are migration tools and manuals provided by IBM [I], Borland, Sun Microsystems, etc., explaining and helping with the passage from their own (old) technologies, or some other company's (current) technology, to their own current technologies.

There is more in the area of migrating legacy systems [B] when the emphasis is on understanding the system's data model and reexpressing it. This is data migration rather than source migration, and raises the additional problem (which we don't consider here) of translating existing application data to a new format and schema.

The work of Chris Verhoef and his collaborators is exceptional, being well-written and bang on the target of source migration. Much of the experience and ideas of the present paper are reflected very closely in Verhoef's work, especially [T], [S], and [V].[1]

The author's own experience in this area was established 1995 – 2000 while working at Legasys Corp. (Kingston). Automatic dialect conversion tools were made, maintained, and used, for COBOL (4 dialects ), PL/I (2 dialects) and RPG (3 dialects). Semi-automated API migration and language conversion projects were piloted in 1999–2000 including COBOL-Java, RPG-COBOL, AS/400-MQSeries, and SQL-SQLj. None of the numerous working papers written during that time is published or available. See the list of **Acknowledgements** below.

**Kinds of Source Migration**. At Legasys we identified three kinds of source migration task: in order of difficulty and risk they are:

- dialect conversion
- API migration
- language migration

When planning a source migration it's important to recognize which kind of task it is. The techniques which are needed differ in each case, and so do the risks.

**Dialect Conversion** occurs when the supporting compiler technology changes, either to a new version of the compiler or to a new compiler family. An example of the first kind is migration to COBOL/370 from VS COBOL, to Java 2 from Java, or from GNU C++ V2.93 to (the new) V3.0. Generally speaking successive compiler versions are meant to be successively compatible, even when new features are added. However, there is usually an impact on large existing code bases, and for this reason code owners are reluctant to migrate even to a new compiler version until the need arises for some other reason (such as access to new features or obsolescence of old ones).

As an example, here's a little piece of Borland C++ (from the SORTIE code base [C]) which I have to convert to GNU C++ in order to perform low-level design recovery:

```
#pragma resource "*.dfm"

class TYesNoForm : public TForm
{
__published:    // IDE-managed Components
        TLabel *Message;
        TPanel *Panel1;
…

void __fastcall TTreeMapFm::AddClick(TObject *Sender)
{
    AnsiString theCaption;
    AnsiString theMessage;

   if(atoi(XCoord->Text.c_str()) >= atoi(PlotlenXEdit->Text.c_str()))
    {
      theCaption = "Tree Map File Warning";
…
```

Here we can see various extensions which are not part of ANSI or GNU C++. The simplest keyword substitutions (`__published`, `__fastcall`) can be converted by a stream editor. Compiler directives

[1] My access to Verhoef's writings is strictly from the Web. Frustratingly, he doesn't usually provide conventional references, even for conventionally published stuff.

(#pragma) often cannot. The third and most complex issue here is the use of AnsiString in BCB instead of string in standard C++. This is a typical case where the *idiom* in the target dialect is different for some kind of data or control structure. One may proceeed by:

- blind substitution, fix up later
- implement the source idiom in the target environment (e.g. recreate some or all of AnsiString in gcc).
- API migration, see below

The key to dialect conversion, making it the simplest kind of migration, is that it can proceed iteratively, migrating one module at a time or one feature at a time, with testing and integration according to the procedures already in place.

A possible translation of the above slice is:

```
class TYesNoForm : public TForm
{
public:          // IDE-managed Components
         TLabel *Message;
         TPanel *Panel1;
…

void TTreeMapFm::AddClick(TObject *Sender)
{
   string theCaption;
   string theMessage;

   if(atoi(XCoord->Text.data()) >= atoi(PlotlenXEdit->Text.data()))
   {
     theCaption = "Tree Map File Warning";
…
```

In the above transcription, the string operations have been substituted directly. However, the interface to AnsiString (which is a calque Pascal long strings) is not the same as the string class of ANSI C++. Conversion often involves **API Migration**, which is replacement of dependence in source code on some external API. This may be as simple as a dialect-specific data type (as in this example) or as complex as database or job control access languages, as in the following example (from Oracle [O]). This sequence of code depends on the JDBC API for access to relational databases:

```
import java.sql.*; // Package for JDBC classes
…


public void selectRecords(String p_code,String p_name,String p_partner) {
   // The query for selecting from the airlines table
   String l_query = "select * from airlines "
                    +"where code like ? AND name like ? AND partner like ?";
   try {
       // Create a PreparedStatement based on the query in l_query
        PreparedStatement l_pst = m_connection.prepareStatement(l_query);
        // Bind the PreparedStatement with corresponding values
       l_pst.setString(1,p_code);
       l_pst.setString(2,p_name);
       l_pst.setString(3,p_partner);

        // Execute the PreparedStatement
        ResultSet l_resultSet = l_pst.executeQuery();

        // Populate the Result set, retrieve rows
       while (l_resultSet.next()) {
           String l_code = l_resultSet.getString(1);
… …
```

To convert this to use SQLj, with accompanying type checking for embedded SQL, and more conciseness, is a typical API migration task. Affected regions of source code may have to be rewritten, but large areas will be quite unaffected (e.g. processing business rules, user interface, etc.) The architecture of the application as a whole will not change very much. On the other hand, as can be seen by the SQLj

version:

```
//SQLJ runtime classes
import sqlj.runtime.*;
import sqlj.runtime.ref.*;
…
public void selectRecords(String p_code,String p_name,String p_partner) {
   //The query for selecting from the airlines table
   SelRowIter l_selRow = null;

   try {
       #sql l_selRow  = { select * from  airlines
                         where code like :p_code
                         AND   name like :p_name
                         AND   partner like :p_partner
                         };




        // Populate the iterator and process all rows returned
       while(l_selRow.next()) {
           String l_code = l_selRow.code();
… …
```

it is necessary to perform some local program understanding (design analysis) in order to discover which patterns are subject to transformation [R].

In API migration it's still possible to rely on integration testing and iterative reengineering, albeit to a lesser extent, to mitigate the risk of large-scale and wide spectrum change.

The most risky type of source migration, attempted (and failed) quite frequently for all that [T]) is **Language Migration**. In this case the decision is taken to convert to a wholly new programming language. It is particularly difficult, just as translation of human language is difficult, because three issues:of the need to reexpress everything: and t

- everything has to be reexpressed
- similar constructs have different meanings
- source code contains many factors [M] all of which express levels of meaning (documentation, software architecture, system architecutre[2] , functionality) all of which may need to be translated and woven together

- Consider the following swatch of COBOL intended for conversion fo Java (a real task!)

```
PROGRAM-ID.  AR500X.
…
FILE-CONTROL.
    SELECT AR500D ASSIGN TO MAIN-FILE.

FILE SECTION.
FD  MAIN-FILE RECORD CONTAINS 80.
01  MAIN-BUFFER PIC X(80).
…

WORKING-STORAGE SECTION.
01  MAIN-RECORD.
    COPY 'AR500DC'.
77  END-OF-FILE PIC X.
…
PROCEDURE DIVISION.
    PERFORM OPEN-FILES
    PERFORM MAIN-LOOP UNTIL END-OF-FILE.
    PERFORM CLOSE-FILE.
    GOBACK.
```

[2] Software architecture is to system architecture as proofs are to theorems.

```
        OPEN-FILES.
            OPEN INPUT MAIN-FILE.
            …
```

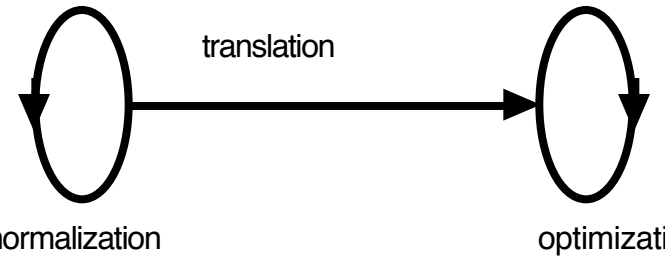
- Even a naïve translation requires typical decisions such as: what should be the main routine? What corresponds to a program? What corresponds to data aliasing (an essential idiom in COBOL, meaningless in Java)? What sould the source architecture be, corresponding to structures of COPY files? Some decisions are recorded in this tentative translation (which results from a formal process):
-

```
import AR500.*;
class AR500X {
…
            File mainFile = new File ("AR500D");

…
…
            String mainBuffer = new String (80);
…

…
            Ar500.Card mainRecord = new Ar500.Card ();
            boolean endOfFile;
…
        void mainProcedure () {
            openFiles ();
            mainLoop ();
             while (!endOfFile) mainLoop ();
            closeFile ();
        }

        void openFiles () {
             mainFile. open (mainBuffer, 80);
            …
```

**Phases of Source Migration**. In order to keep the difficulties under control, there are three phases of source migration: normalization, translation, and optimization. These phases are

> **Normalization** is reducing the translation space. At the lexical level, some kind of source factoring [M] is essential so that the “code factor” is available to design analysis. Normalization is in fact *directed dialect conversion*, where the target dialect is chosen to make the translation step easier. It may involve API migration as well, if there are APIs which will not be available on the target environment.
>
> **Translation** is actually the hardest step! The goal is to make a version which runs in the target environment and can be improved to a maintainable artifact by optimiization. The “semantics should be preserved” of all factors: this is difficult to define as a requirement. We have found that “blind translation” is the most effective approach, since there is very little support semantically for the intermediate stages.
>
> **Optimization** is similar to optimization in compiler back-ends, except that the improvements intended are improvements to reflect the goals of migration, especially “make maintainable source”. This will probably involve removing or simplifying APIs inherited from the source during normalization, recognizing and improving the idioms to conform to “native style” on the target side, and possibly improving the software architecture. This is also *directed dialect conversion*, with a different direction.

The above can be summarised by the following diagram




which because of its resemblance to a weight-lifting device we have taken to calling the **Barbell Model**. It reminds that, in order to keep *balanced* the considerable weight of work required when migrating software source, reengineering tasks should be kept on the ends (where they can be carried out in a single semantic framework or platform, with integration testing and software maangement processes in place) and the tricky task of keeping the ends together is kept narrow.

**Acknowledgements**. All my experience of software migration projects was gained in collaboration with J. R. Cordy, K. A. Schneider, and T. R. Dean, at Legasys Corp. (Kingston). Cordy frequently emphasized the relationship between migration and compilation, and proved it by implementing the back end of the RPG-COBOL translation as a code generator. Schneider suggested a structure for the COBOL dialect conversion process which proved very useful; and also started the use of “barbell” to describe the diagram which we had drawn during one of many valuable discussions. Dean directed our exploration of a very different (“extreme”) approach to COBOL-Java translation, which taught us all a lot about the semantic difficulties we were facing.

Verhoef gives a diagram [T] very similar to the barbell diagram above.

**References.**


[B] *Migrating Lagacy Systems*, M Brodie, M Stonebreaker. Morgan Kaufmann, 1995

[C] *SORTIE*, C. Canham. The project page is http://www.csc.uvic.ca/~sachen/overview.htm

[F] "A Fortran to C Converter", S Feldman, D Gay, M Maimone, N Schryer, AT&T Technical Report 149, 1993

[G] p2c, a Pascal to C Converter. D Gillespie. Search for "p2c" on any hpux mirror such as hpux.ee.ualberta.ca.

[I] *COBOL … Migration Guide*, IBM Corporation, document GC26-4764-04. Try as400bks.rochester.ibm.com/cgi-bin/bookmgr/BOOKS/IGYMG201/CONTENTS.

[K] "Code Migration Through Transformations: An Experience Report", K Kontogiannis, J Martin, K Wong, R Gregory, H Muller, J Mylopoulos. In Proceedings of CASCON'98, Toronto ON. December 1998

[L] *Software Maintenance Management*, B P Leintz, E B Swanson. Addison-Wesley, 1980.

[M] "Processing Software Source Text in Automated Design Recovery and Transformation", A.J. Malton, K.A. Schneider, J.R. Cordy, T.R. Dean, D. Cousineau and J. Reynolds, Proc. IWPC 2001 - IEEE 9th International Workshop on Program Comprehension, Toronto, May 2001, pp. 127-134.

[O] (example source code), Oracle Corporation. (from web site, URL to be supplied)

[R] "HSML: Design Directed Source Code Hot Spots", J R Cordy, K A Schneider, T R Dean and A J Malton. Proc. IWPC 2001 - IEEE 9th International Workshop on Program Comprehension, Toronto, May 2001, pp. 145-154.

[S] "An Architecture for Automated Software Maintenance", A Sellink and C Verhoef. Seventh International Workshop on Program Comprehension, 1999.

[T] "The Realities of Language Conversions", A.A. Terekhov, C. Verhoef. Surf to adam.wins.uva.nl/~x/cnv/.

[V] "Current Parsing Techniques in Software Renovation Considered Harmful", M vdBrand, A Sellink , C Verhoef. From adam.wins.uva.nl/~x/ref/

[W] "A Semantic Foundation for Architectural Reengineering", S G Woods, S J Carrière, R Kazman, Proceedings of ICSM99, 1999, p391-398.

[X] "Virtual PC", Connectix Corp. Visit www.connectix.com.