The Realities of Language Conversions

Billions of lines written in Cobol, PL/I, and other mature languages are still in active use. Many developers have tried to convert these languages to more modern ones, but few have succeeded. This article sheds light on the realities of language conversions and discusses the possibilities and limitations of automated language converters.

Andrey A. Terekhov, *St. Petersburg State University*
Chris Verhoef, *Free University of Amsterdam*

The most influential cost drivers of software engineering relate to management, personnel, and team capability—not software tools, although software vendors and academic researchers emphasize them. Many managers become victims of quack software modification tools and services. This mechanism, also known as the silver-bullet syndrome, is what anthropologists call *name magic*: you just say the name of the thing—"Cobol to Java"—and you have its full power at your disposal.

As the term *name magic* indicates, you don't need proof to support your claims. Anyone desperate for solutions will wholeheartedly accept those unsupported claims.[1]

In fact, many software industry decision-makers find themselves trapped in huge amounts of legacy code, in dire need of modification. At the same time, software engineering educators deliver people skilled in contemporary development rather than enhancement programming, let alone geriatric care for aging legacy applications. Capers Jones[2] even thinks that this phenomenon is one of the 60 most important risks in software engineering. You do not have to be a rocket scientist to figure out that a language converter could solve your personnel problems: a language conversion is supposed to bridge the gap between available knowledge of people and the knowledge needed to solve legacy problems. Is it that easy? This article sheds light on the realities of language conversions; it also provides simple examples that expose the problems—anyone involved in software engineering can understand them.

So How Hard Is Language Conversion?

We encountered a company stating about a Cobol–to–Visual Basic converter, "The converter runs as a simple wizard: it is intended to be something a secretary can run." Furthermore, at a panel of the International Conference on Software Maintenance, a researcher discussing the next century's challenges implied that automated conversion is solved and that the next challenge is paradigm-shift conversion. This means not only converting Cobol to C++ but also text-based procedural code into Web-enabled object-oriented C++. From such observations, we

0740-7459/00/$10.00 © 2000 IEEE

The problem statement for language conversion is simple: convert this system to that language without changing the external behavior.

could conclude that automated language conversions are easy. And let's face it, converting the Cobol calculation `ADD 1.005 TO A` to its equivalent in VB, `A = A + 1.005`, is indeed very easy. Any simple tool can do it.

On the other hand, language conversion seems to be risky business. We know of three companies that went bankrupt and two departments of large software-intensive enterprises that were dismantled, all because of failed language conversion projects. Tom Holmes from Reasoning states that if success means making a profit, then most conversion projects are not successful. A reader of a preliminary version of this article told us of an enterprise that spent $50 million on failed language conversion projects. In his book on computing calamities, Robert Glass mentions the failure of a translation system that would convert software from an obsolete system to a new one. Management told the developers that the conversion problem was limited in scope and that they would need only to convert a limited set of constructs. This turned out not to be true. The postmortem analysis indicated that the converter was perhaps 10 times more complicated than expected; suddenly, what had been technically feasible became economically and technically infeasible.[3]

In the news group alt.folklore.computers, S.C. Sprong, who ported software from Fortran to C, stated, "Low-level porting, even with provided documentation, is one of the blackest software arts; knowing how to do it well will surely get you a first-class ticket to hell."

Needless to say, there is a lot of disagreement on the complexity of language conversion. Perhaps it is theoretically possible to automatically improve a software system's structure to achieve a paradigm shift—for example, by introducing OO concepts. However, this is very difficult to automate and it implies a lot of human intervention.[4] However, the payoff is commensurately large, especially for systems with indefinite life spans. Due to automation problems, most conversion tools apply the technology of syntactic conversion. Even with this seemingly simple and low-level approach, many difficulties occur, and the scale of those intricacies is not yet fully understood.

Despite the lack of solid publications about language conversions (we listed the most useful references,[5–13] but some are difficult to find or are in German, so contact us for copies or pointers), lots of vendors advertise migration tools and services. Many companies say they can convert your systems to whatever language you want. Although some of them may be doing a great job, others who claim to have technology and skills are not. For instance, one company advertising on the Internet provided examples of converted code that appeared not even to be compilable. Another company claimed to be able to convert PowerBuilder to Java, but after our inquiries they had neither the experience nor the tools to do the job. Instead, they were claiming to have a process! In a quote (which we translated from German to English) from his book on OO software migration, Harry Sneed summarizes the state of the practice in the transformation marketplace as follows:[12] "The reality looks different. Those who can read between the lines recognize that the problems are grossly simplified and that the advertised products are far from being ripe for use in practice."

Requirements for Language Conversion Aids

The problem statement for language conversion is simple: convert this system to that language without changing the external behavior. From an abstract point of view, a language migration project seems deceptively simple. Consequently, requirements for language converters are often not formulated.

Usually, the availability of constructions that facilitate the expression of a solution determines how easy it is to formulate a solution for a particular problem. We could call such elements in a programming language *native language constructs*. For instance, if we wish to express a conditional problem, a language that supports a conditional language construct is more convenient than a language lacking an if-then-else construct. If we must use the latter language, we must simulate the conditional construct. Such code fragments are called *simulated language constructs*. We can, for instance, simulate object orientation in a language lacking OO support.

The language conversion problem for a serious software system amounts to mapping its native and simulated constructs to hopefully only native constructs available in the target

language(s), as Figure 1 shows. There are at least six possible categories in this mapping (expressed as the six arrows in the figure), all of which we have encountered. Requirement specifications usually mention only the native-to-native part of the mapping, in the form of a statement-by-statement conversion. This phenomenon is in accordance with people's tendency to focus first on the easiest problems. One of us (Chris Verhoef) was an external reviewer of several large-scale language conversion projects. Most of them failed because the hard problems were avoided in the requirements. In one case, for instance, about 80% of the requirements specification dealt with the graphical user interface, while the actual language converter was represented by a single arrow. Someone underestimated the problem and thus omitted the hard parts from the requirements.

Underestimation often leads to runaway projects. The first project failure delays the actual software conversion, which in turn triggers increased pressure on the next development team to deliver the converter. This pressure makes it easy for the new team to skip requirements altogether; after a few failure cycles, total breakdown occurs, clarifying the demise of the earlier-mentioned language conversion companies and departments.

To develop a source-to-source converter, you need at a minimum several requirement specifications:

- You must inventory the native and simulated language constructs of the system that needs conversion.
- You must develop a conversion strategy for each language construct. Specifically, you must list the input and output fragments that describe the converter's desired behavior.
- You must make an explicit statement about whether the converted system should be functionally equivalent to the original system. Intuitively, you would think this is always so, but in practice, a sophisticated automated modification effort usually exposes faults and unsafe code in the original system. Often, the customer then requires the developers to fix these faults as well. Thus, potential requirements creep must be dealt with in the requirements. Note that it also hampers testing the new system because regression testing is based on equivalence.
- You must include a statement about whether the original system's test sets are to be converted. If errors in the tests are exposed, the requirements must state the policy toward modification.
- You must set as your goal maximum automation of the conversion (enabling minimal human interference).
- If you plan to maintain the converted system, you must make it maintainable. For instance, if the original maintenance team is going to continue in its role, the new system should be as similar as possible to the original system so that the team can recognize the original code. If, on the other hand, the maintenance team is new, the conversion should try to use the target language's idiom so that maintainers recognize the code as normal for that language. In other cases, the original source code (say, in Cobol) is used for modifications even after a successful conversion (say, to Java), because the translation process is so automated that it's easier to re-generate final Java code than to make changes in Java. This also helps maintainers if they are familiar with the original language but not yet with the target one.
- You must make the converted system's efficiency adequate both in compilation and execution time.
- If you will use the converter many times, the time required for conversion is relevant. It is not always feasible to optimize this without distributing calculations over many machines.
- If you plan to maintain the converted system, its size must not exceed the original system's size by much (if at all).

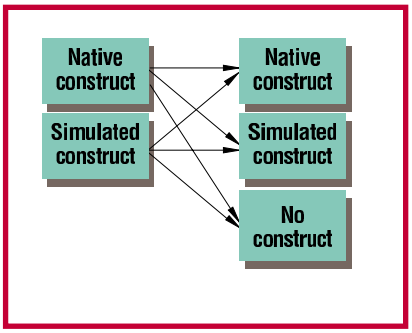

Figure 1. A mapping of constructions between languages.

Apart from these explicit requirements, there is always an implicit understanding of how the converter will work. This reflects the customer's expectations of the advantages associated with transferring the system to a more contemporary environment. These imaginary advantages often motivate the conversion process but they are rarely realized. A popular misconception is that after conversion the system is change-enabled so that totally new features can be imple-

Although we do not always realize it, programming languages usually have idiosyncratic data type conventions.

mented easily. The problem is aggravated by companies marketing conversion software as yet another silver bullet; the quality of such converters is often less than optimal, and in some cases even nonexistent.

An automatically converted program is usually not as good as a new one developed within the full range of a contemporary programming language. Although the outcome of conversion should ideally be code that acts as if it is written in the target language and uses the target language's features and idiom, actual converted programs often retain the idiom of the source language (more on this later). Some people expect that the structure of the program will surely improve after automated conversion, but conceptual changes to the application will always remain labor-intensive and require human interference.[5,13] For instance, imagine replacing mainframe CICS with C++ using Microsoft Transaction Server.

Technical Problems

A list of input and output patterns is very helpful when converting from one language to another—in fact, this is the hard part of language conversion.

Converting data types

One of the first problems is converting data types. Although we do not always realize it, programming languages usually have idiosyncratic data type conventions. For instance, many people consider C++ and Java to be similar, so a native-to-native conversion seems a simple task. Yet, their data types reveal many differences. For example, C++ has pointer-type variables but Java doesn't; Java has Booleans but C++ doesn't. Also, C++ data type sizes vary from platform to platform, whereas they are fixed in Java. So even when converting between C++ and Java, we run immediately into the problem of representing idiosyncratic data types.

Thus, you should not be surprised that the differences between languages like Cobol or PL/I and languages such as Java, VB, or C++ are perhaps insurmountable. For example, consider this PL/I data type:

```
DECLARE C FIXED DECIMAL (4, -1);
```

The variable `C` occupies three bytes, with the decimal point assumed to be one position to the right of the number. Thus, `C` may contain values 123450 and 123460 but not 123456. `C` ranges from –99999*10 to 99999*10, and all values assigned will be truncated at the last digit, so the assignment `C = 123456` is equivalent to `C = 123450`. Since the last digit is always a zero, it is not stored at all. Clearly, neither this data type nor the assignment operator correspond to any standard C++ data type and assignment operation. (The article by Kostas Kontogiannis and colleagues[5] converts a PL/I dialect to C++ but does not address this problem.)

Cobol to Visual Basic

In a Cobol-to-VB conversion, one possibility is to convert only the data types that have an equivalent in the target language. This is the native-to-native mapping shown in Figure 1. The converter would report variables of all other data types to the user, suggesting that the user rewrite the parts of the code that use them. This simple Cobol code fragment illustrates the problems with data type conversions:

```
DATA DIVISION.
 01 A PIC S9V9999 VALUE -1.
PROCEDURE DIVISION.
 ADD 1.005 TO A.
 DISPLAY A.
```

The variable `A` can represent values such as –3,1415; here, it initially represents –1. In the procedural part of the program, we add the number `1.005` to `A`, then we print the result to the screen. A converter could turn this simple Cobol program into the following VB program:

```
Dim A As Double
A = -1
A = A + 1.005
MsgBox A
```

The VB program declares the same variable `A` as a `Double`. The variable `A` is initialized and obtains the value `–1`. VB represents the Cobol `ADD` as a VB + operator. When we run both programs, the VB code yields a different result (`+0.0049`, indicating a rounding error) than the Cobol code.

The reason for the small difference is the fact that the Cobol code uses a fixed-point data type and the VB fragment uses a floating-point data type. A number of reference books have documented the poor precision of VB's floating-point implementation.

For the fragment just given, we can solve the rounding problem by using a VB data type called `Currency`. But consider a slightly different Cobol program:

```
DATA DIVISION.
 01 A PIC S9V99999 VALUE -1.
PROCEDURE DIVISION.
 ADD 1.00005 TO A.
 DISPLAY A.
```

This is now converted using the `Currency` data type:

```
Dim A As Currency
A = -1
A = A + 1.00005
MsgBox A
```

The Cobol code calculates with more precision than the previous one, yielding the expected result. However, the VB code prints `0.0001`, which is twice as much as in the second Cobol program. This problem occurs because the `Currency` data type uses four-digit precision and rounded results for smaller amounts. So the `Currency` data type will misbehave in other contexts.

So, no single data type in VB can handle the fixed-length record structure in Cobol. Therefore, *any* simple strategy to convert these Cobol data types is doomed to fail, because different contexts require different solutions. Clearly, a sophisticated data type analysis is mandatory even in the simple example programs shown here.

Cobol to C

We mailed the simple Cobol program just listed and its conversion to VB to the researcher mentioned earlier who had listed the next century's 10 challenges—to check whether he considered this a syntactic transformation. He confirmed this and stated, "I contrast syntactic transformation (which we know how to do well) with deeper ones." He did not observe the rounding problem in the VB conversion. This is perfectly understandable; he told us later that he had never worked with Cobol or VB, nor on a converter between the two. He also converted our first Cobol example into C:

```
double A = -1.0;
A =+ 1.005;
printf("%d\n", A);
```

This code cannot be compiled at all. We think he meant to include some `main` function:

```
double A = -1.0;
main ( ) {
  A =+ 1.005;
  printf("%d\n", A);
}
```

This code compiles and produces `1072698490`. The format string `%d` does not expect floating numbers, hence the erroneous output. (At least we now know the answer to the question "Why does 2 + 2 = 5986?" on the cover page of a book on practical C programming.[14]) It should be another format string, `%f`. After this repair, the code produces `1.005000`. Of course, the `=+` should be `+=`. After another editing session, we obtain `0.005000`. More editing to silence all the compiler warnings leads to the following neat C code:

```
#include <stdio.h>
double A = -1.0;
int main (void) {
  A += 1.005;
  printf("%f\n", A);
  return (0);
}
```

Unfortunately, this program is still wrong: although we have the correct arithmetic answer, the Cobol program's output differs from the C program's output. We did not mention this before, since we want to address one problem at a time. The actual Cobol output is `00050+`. The literal output of the C program is `0.005000`. Obviously, the two programs are not semantically equivalent. This may seem nit-picking, but what if this Cobol code were part of a successful multilingual invoice layout system where printing in the wrong format would have devastating consequences? After all, only business-critical systems are candidates for conversion. In our next attempt to convert the Cobol program, we come up with the following:

```
#include <stdio.h>
double A = -1.0;
void display (double A) {
     char s[16];
     sprintf(s, "%+#6.4f", A);
     printf("%c%s%c\n",
       *(s + 1), s + 3, *s);
```

No single data type in VB can handle the fixed-length record structure in Cobol.

We hope you will doubt whether our C program is indeed equivalent to the Cobol program.

```
}
int main (void) {
     A = A + 1.005;
     display(A);
     return (0);
}
```

First, observe that this program has little resemblance to the initial C solution. Second, when we imagine that the `display` function is in a library, the code shows strong resemblance with the Cobol source. Thus the idiom of Cobol is entirely preserved in the C code.

Let's discuss the code just given. We see the initial stage of a library function called `display` in the code that emulates the output behavior of the Cobol `DISPLAY` statement. We declare a character string of length 16 and put the value of `A` in a buffer. Then we use the `printf` function to format it in the correct way: we skip the sign and print the contents of the pointer to the second array value using `*(s + 1)`, then we print the contents of everything behind the dot using `s + 3`, and finally we print the contents of the first array value, which is the sign using `*s`. Still, we cannot be sure that this C program is a correct conversion of the Cobol program. For instance, what is the exit status of the original Cobol program when executed on a mainframe? Does it equal 0, the exit status of our C program? Or, is the level of compiler warnings the same for the original and converted programs? If not, maybe the converted program is not accepted in the production environment; who knows? But more importantly, we hope you will doubt whether our C program is equivalent to the Cobol program. Indeed, doubt is healthy when it comes to converted code.

Cobol to Java

Any language construct that can be abused will be abused. These so-called clever uses of programming language constructs can lead to unexpected discrepancies between the input–output behavior of the original code and the converted code. There are problems associated with overflow, type casting, and other complicated type manipulations. Consider the following Cobol program.

```
DATA DIVISION.
  01 A PIC 99.
PROCEDURE DIVISION.
  MOVE 100 TO A.
  DISPLAY A.
```

In this fragment, a variable `A` is declared that can represent two digits like 42. In the procedural code, a constant that is too large for this data type is assigned to `A`. Then we print the result. A converter could transform this code into the following Java fragment:

```
public class Test {
 public static void main(String
 arg[]) {
    short A = 100;
    System.out.println (A);
 }
}
```

We declare a short integer variable `A` and assign the value 100 to it. Then we print the result. Of course, both programs yield entirely different output values: the Cobol program prints `00`, the Java program displays `100`.

Solving the problems of unexpected side effects with data types is one of the realities of language conversions. To fight these problems, we need a different approach, which we call *data type emulation*.[7] For each source language data type that does not have a precise equivalent in the target language, we create dedicated support emulating the transactions specific to this data type. So, in a sense, we add some constructs to the target language so that the "Native" arrow in Figure 1 moves from "No construct" to "Simulated construct." For example, the Cobol variable declarations

```
01 A PIC 9V9999.
01 B PIC X(15).
```

do not have a satisfactory equivalent in, say, Java. Therefore, their conversion could have the following syntax:

```
Picture A = new Picture
  ("9V9999");
Picture B = new Picture
  ("X(15)");
```

where the `Picture` class emulates in detail the source data type behavior, dealing with the treatment of assignments, conversion to related data types, and overflow handling. Obviously, the converted program has a strong Cobol flavor, although it is a Java program. You could call such programs Java-compliant Cobol programs, but maybe not Java programs (we already saw a C-

(a)

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TEST-1.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 TMP PIC X(8).
01 H-DATE.
   02 H-MM    PIC XX.
   02 FILLER PIC  X.
   02 H-DD    PIC XX.
   02 FILLER PIC  X.
   02 H-YY    PIC XX.
PROCEDURE DIVISION.
PAR-1.
  MOVE CURRENT-DATE TO TMP
  MOVE TMP TO H-DATE
  DISPLAY 'DAY   = ' H-DD.
  DISPLAY 'MONTH = ' H-MM.
  DISPLAY 'YEAR  = ' H-YY.
```

(b)

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TEST-2.
DATA DIVISION.
WORKING-STORAGE SECTION.
  01 TMP PIC X(6).
01 H-DATE.
   02 H-MM    PIC XX.
   02 FILLER PIC  X.
   02 H-DD    PIC XX.
   02 FILLER PIC  X.
   02 H-YY    PIC XX.
PROCEDURE DIVISION.
PAR-1.
  ACCEPT TMP FROM DATE
  MOVE TMP TO H-DATE
  DISPLAY 'DAY   = ' H-DD.
  DISPLAY 'MONTH = ' H-MM.
  DISPLAY 'YEAR  = ' H-YY.
```

Figure 2. Two seemingly equivalent Cobol programs: (a) OS/VS Cobol and (b) VS Cobol II.

compliant Cobol program). As soon as we start to emulate data types, we also have to question the compositionality issue. That is, if we use emulated data types in a native arithmetic construct of a converted program, is the result correct? If not, we have to create special functions like `Add(Pic, Pic)` or create special methods implementing the arithmetic operations that give the correct behavior for the emulated data types, for instance, `a.Add(b)`. In C++ we can overload operators like `+`, `-`, `*`, and `=`. Now imagine a programmer who is adding functionality to this C++ program, and the wrong emulated, overloaded + is applied instead of the usual operation. Given all these problems, we find it hard to believe that converted programs are more maintainable, change-enabled, contemporary, component-based, or any other qualification that is often heard as primary reason for a language conversion.

OS/VS Cobol to VS Cobol II

We have seen that it is not so easy to safely convert the data part of programs to a target language. Converting the procedural code is not easy, either. So let us limit our expectations a little and look at dialect conversion. As an example, we will focus on converting from a Cobol 74 dialect called OS/VS Cobol to a modern Cobol 85 dialect called VS Cobol II. The compilers are both from IBM, and the code runs on IBM mainframes. Many of us think that such conversions are a non-issue; let's see. The following code excerpt displays the word `IEEE`:

```
PIC A X(5) RIGHT JUSTIFIED
   VALUE 'IEEE'.
DISPLAY A.
```

This is valid syntax in both Cobol dialects, so it is tempting to assume conversion is unnecessary. The problem is that the *same* syntax can have *different* behavior. For instance, the OS/VS Cobol compiler prints the expected result, namely "` IEEE`", which is right-justified. The VS Cobol II compiler displays the output "`IEEE `", which is *left*-justified (if you wonder why, we have two words for you: ANSI standards). This is not an isolated case.[15-17] Carl Gehr and Rex Widmer call this "same syntax, different behavior" in their *Cobol Migration Course*.[17] We call it the homonym problem.

Another case of a problem that is not easily detected is presented in Figure 2.[18] Basically, the OS/VS Cobol program in Figure 2a declares a `TMP` variable with eight positions and then defines a data type `H-DATE` for dealing with dates like 13/01/99. In the procedural part, the special register `CURRENT-DATE` stores today's date in the `TMP` variable. The program stores this value in the simulated date field, then prints the day, month, and year. To convert this program to a newer Cobol dialect, we must replace the `CURRENT-DATE` special register by the new system call `DATE`. The type of `DATE` is YYMMDD, which is not the same as DD/MM/YY, the type of `CURRENT-DATE`. IBM's *Cobol/370 Migration Guide*[19] proposes converting the type of variable `TMP` from `PIC X(8)` to `X(6)` and converting `MOVE CURRENT-DATE` into an `ACCEPT`

Procedural code conversion is tightly coupled with the conversion of the data types involved in the procedural code.

statement. This leads to the VS Cobol II program in Figure 2b.

The IBM solution breaks down in this context, because the type of the variable TMP assumes in the MOVE statement that it equals the type of the variable H-DATE. But this is no longer true, so the converted program's output is completely erroneous. Executing the converted program on 15 September 1999 results in the output

```
DAY   = 91
MONTH = 99
YEAR  =
```

The program has mapped the string 990915 onto H-DATE. The subfield H-MM obtains the first two digits and is set to 99, then the FILLER is set to 0. The subfield H-DD gets the next two digits, 91; the next FILLER gobbles up the last 5; and H-YY gets nothing. So, the program prints first the contents of H-DD as value 91, then H-MM as 99, and H-YY as nothing.

How do we fix this? One solution could be to also convert the H-DATE field and all code that depends on it. This solution ripples through the program, and it can also affect other programs: the data type could be used in a database or passed as a parameter in a call to another program. Procedural code conversion is tightly coupled with the conversion of the data types involved in the procedural code. If we follow the IBM solution, we must modify the entire system. By using data type emulation, we can prevent the ripple effect, namely by converting to the following code:[18]

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TEST-3.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 F-DATE.
   02 F-YY PIC XX.
   02 F-MM PIC XX.
   02 F-DD PIC XX.
01 TMP PIC X(8).
01 H-DATE.
   02 H-MM   PIC XX.
   02 FILLER PIC  X.
   02 H-DD   PIC XX.
   02 FILLER PIC  X.
   02 H-YY   PIC XX.
PROCEDURE DIVISION.
PAR-1.
  ACCEPT F-DATE FROM DATE
  STRING F-MM '/' F-DD '/' F-YY
    DELIMITED SIZE INTO TMP
  END-STRING.
  MOVE TMP TO H-DATE
  DISPLAY 'DAY   = ' H-DD.
  DISPLAY 'MONTH = ' H-MM.
  DISPLAY 'YEAR  = ' H-YY.
```

First, we declare a fresh variable F-DATE with the same type as the procedural code's new system call. We store the result of the DATE system call in the fresh variable F-DATE, then we emulate the old special register by storing the old data type in TMP. Now all the other code that relies on the old date format runs as if we were still using the old special register. The entire ripple effect is prevented, and the solution can be 100% automated, though it may not be as beautiful as it could be. IBM has corrected this error in newer versions of their migration guide.

Turbo Pascal to Java

The following Turbo Pascal program is based on code written in a proprietary language that needed conversion to Java. We have transposed the problem to Turbo Pascal to make the code compilable for others.

```
Program StringTest;
var s: string;
 a: integer;
begin
 s := 'abc';
 a := pos ('d', s);
 writeln (a);
 s [pos ('a', s)] := 'd';
 writeln (s);
end.
```

We have two global variables. First, we set the string variable to abc. Then we set variable a to the offset of the first occurrence of the string d in the string abc. Since there is no such occurrence, the value 0 is assigned to a. We write the result to the screen. Then on the first occurrence of substring a in the string abc, we replace a with d, and we write the string to the screen. So the program's output is 0 and dbc. The following Java program could be the output of a naïve, automatic source-to-source converter:

```
public class StringTest {
 public static void main
```

```
    (String args[]) {
  StringBuffer s =
    new StringBuffer ("abc");
  int a;
  a = s.toString().indexOf('d');
  System.out.println (a);
  s.setCharAt (s.toString()
    .indexOf('a'), 'd');
  System.out.println (s);
 }
}
```

Here, we declare `s` to be `abc` and we set `a` to the offset of the first occurrence of `d` in the string `s`. However, the convention in Java is that when this does not occur, the return code equals `-1`, so `-1` is printed on the screen. Thus, the semantics of the first part of the Java program is not correct. Fortunately, the second part is converted correctly. We take the return-code side effect into account and convert the Pascal program as follows:

```
public class StringTest {
 public static void main
    (String args[]) {
  StringBuffer s =
    new StringBuffer ("abc");
  int a;
  a = s.toString()
    .indexOf('d') + 1;
  System.out.println (a);
  s.setCharAt (s.toString()
    .indexOf('a') + 1, 'd');
  System.out.println (s);
 }
}
```

To solve the return code problem in the first part, we add 1 to the index to simulate the Turbo Pascal return code. The value of `a` is correctly 0. However, the second part is now incorrect: the output is `adc` instead of `dbc`. Apparently, the code replaced the second position in the string `abc` with a `d`. This is due to another side effect of conversion: arrays in Java start with 0, instead of 1 as in Turbo Pascal. When we take this effect into account, we get

```
public class StringTest {
 public static void main
   (String args[]) {
  StringBuffer s =
   new StringBuffer ("abc");
  int a;
  a = s.toString().indexOf('d')
   + 1;
  System.out.println (a);
  s.setCharAt ((s.toString()
   .indexOf('a') + 1) - 1, 'd');
  System.out.println (s);
 }
}
```

We add one to the `indexOf` function to correct for the difference in return codes. For Java arrays, we subtract 1 from the array counter so that we correct the difference with Turbo Pascal. During the restructuring phase of the converted code, we can normalize for cumulative calculations in the code and restructure the code

```
s.setCharAt ((s.toString()
  .indexOf('a') + 1) - 1, 'd');
```

by rewriting it as

```
s.setCharAt (s.toString()
  .indexOf('a'), 'd');
```

Now we see why in the first conversion attempt the second part gave the correct answer: it was a cumulation of two errors that canceled each other out and provided the right answer by accident. Or, as Scott Adams formulates it in his book *The Dilbert Principle*: Two wrongs make a right, almost.

As Scott Adams formulates it in his book The Dilbert Principle: Two wrongs make a right, almost.

The Gory Details

The essence of reengineering is getting all the gory details right—and the list of annoying issues that must be taken care of is endless.

Special difficulties are caused by operations involving the internal representation of variables as well as other operations interacting with memory. Such cases require separate runtime support procedures. Consider the following example:[20]

```
STRING ID-1 ID-2 DELIMITED BY "*"
       ID-4 ID-5 DELIMITED BY SIZE
  INTO ID-7 WITH POINTER ID-8
  ON OVERFLOW GO TO OFLOW-EXIT.
```

This Cobol sentence combines the partial or complete contents of four different data items into one single data item `ID-7`, also providing the pointer `ID-8` to the last character position in the receiving field. More-

Even when we restrict ourselves to a dialect conversion of two IBM products, we encounter problems.

over, when too many elements are put in `ID-7`, the control flow of the program goes to a paragraph named `OFLOW-EXIT`. The `STRING` statement deals with the internal representation of the variables and must be emulated when converting into a language not supporting this kind of composition of variables. To preserve the program's semantic correctness, the target types ought to have the same internal representation as the source ones.

This problem is closely related to the data type casting problem, and a solution of the first depends on a solution to the second. If the data types are emulated, then the operations must be emulated as well. We do not believe that merely using native data types will lead to satisfactory solutions for Cobol programs containing this kind of operation. This is not the only operation of this type. Cobol contains native syntax for searching in data items, counting of data items, and replacement by other data items. For instance, the `INSPECT` statement specifies that characters (or groups of characters) in a data item are to be tallied, replaced, or both. Automated conversion to languages not supporting this kind of functionality in a native way must be extended with syntax and semantics capturing such constructs, which brings us to the more general issue of a language domain focus. The example just given clearly shows that Cobol contains a rich set of constructions to deal with rather sophisticated data-processing tasks. Any attempt to convert such a typical Cobol construct to a language that has a different domain focus is doomed to fail, for the latter lacks the language support to deal with the other domain. As one reviewer pointed out to us: Analogously, Cobol is pretty hopeless as a systems programming language, and it would be difficult to impossible to translate a good systems program (in any language) into Cobol. The converse of this statement, namely that conversion between similar languages would be easy, is not at all true. The homonym problem is a counter example, which shows that conversion is *always* intricate.

Discussion

The examples we have presented clearly illustrate that language conversions are grossly underestimated—even by well-known reverse-engineering experts.[21] Even when we restrict ourselves to a dialect conversion of two IBM products, we encounter problems. IBM seems to have underestimated the conversion of their own dialects.

The examples clearly illustrate that automated language conversion is much more difficult than many people anticipate. A possible cause of underestimating the problems is that the surface syntax of the converted arithmetic looks deceivingly similar to the original. As with `ADD 1.005 TO A` and `A = A + 1.005`, we are fooled by our perception of arithmetic reasoning. Moreover, the printing routines for Cobol (`DISPLAY`), VB (`MsgBox`), and C (`printf`) also have the same look and feel, but their semantics are not in accordance with our intuitions. Finally, when we restrict ourselves to dialect conversions, the problem becomes even harder: while the programs compile under many compilers, the semantics of the same syntax differ from compiler to compiler. As a consequence, the semantics of converted code will usually differ from the original unless we take many precautions.

The examples also show that it is dangerous to map data types of one language to an approximate data type in the target language (illustrated by the mapping in Figure 1 from "Native construct" to "No construct").

The use of native versus emulated data types depends on the requirements that are most important for the conversion. Native data types do not always lead to correct code, but the problems with data type emulation are numerous and varied as well. Using the native data types of the target language might simplify maintenance (because it delivers less alien code), but it clearly reduces the level of automation or affects the semantic correctness of the result. Applying data type emulation leads to more automation and more correct programs, but at a price of extra work required for writing runtime libraries, more complicated maintenance, and loss of efficiency in the performance of the converted system.

Both techniques can be used simultaneously. It is possible, for instance, to first analyze whether a source program is *type-safe*; that is, we check that problems like the ones we have addressed earlier are not present. If that is the case, we can opt for a translation where we map to approximate native data types in the target language, and when there are problematic parts, we can choose to con-

vert these parts using data type emulation.

Also, the homonym problem reveals a possible misconception that syntactic equality between language constructs of different languages or dialects would be a useful indicator of the complexity of a conversion project—that the more equal the languages are, the more easy a conversion would be. In fact, it is a very nonintuitive measure for complexity of language conversions because the more syntactically equal languages are, the more difficult it becomes to detect whether there are differences. In addition to all the language conversion problems we have, we must also deal with semantic differences that we cannot even detect syntactically.

Any decision maker considering language or dialect conversions to solve a problem should realize that the problems that are perceived to be solved by the conversion will be replaced by other, perhaps more intricate problems. The more code that needs to be converted, the more automation becomes a necessity. This in turn leads to more alien code. If you are lucky, the only connection with object orientation after a major conversion effort is the inheritance of the maintenance problems you already had.

A Process for Conversion

Converting application software from one language to another is usually done to simplify maintenance. Therefore, the target source texts must be well structured, contain as little global data as possible, and so on. Since source language programs rarely comply with these requirements, any sensible language conversion should first start with extensive restructuring—despite the problems you will encounter with the classical restructuring tools.[9]

Certain steps are necessary in a language conversion. The simple example we will use is based on work that was done for a Swiss bank.[11] Figure 3 depicts the basic language conversion process as we experienced it.

First of all, we restructure the original program so that the problematic arrows in Figure 1 are removed as much as possible. For example, Cobol programs do not necessarily have a `main`, but C programs commonly do. Therefore, in converting from Cobol to C, restructuring the Cobol code should include adding a simulated `main`. Subroutines are not common in Cobol but

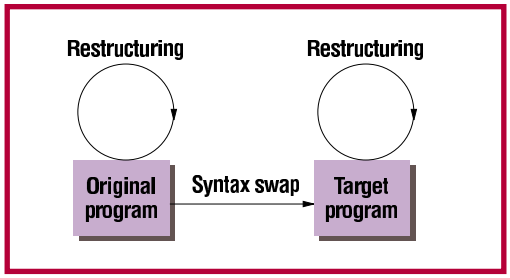

Figure 3. A process for language conversion.

are in other languages. It is hard to recover them, especially with unstructured code. For example, in a Cobol-to-Ada conversion described elsewhere,[4] an abundance of `GO TO`s in the source code (one every 225 lines) hampered subroutine identification.

In most conversions, restructuring is prerequisite to what we call the syntax swap, where we swap the syntax of the precooked, restructured original code with the target syntax. This is a relatively easy step. The swapped programs are usually ugly, so another heavy restructuring phase in the target language is necessary to make the new syntax look as much as possible like native code.

To illustrate the process, we will transform an example Cobol program (composed from actual legacy code from the Swiss bank project) into C. We abstracted from the data type emulation problems to focus on the problems with procedural code. We adapted this code to the domain of traveling so that the program's meaning is easy to follow. The code is of the quality that you might expect to be provided in a real conversion: a `GO TO` every four lines. The original code appears in Figure 4a, the heavily restructured Cobol code in Figure 4b.[11]

The output of both programs is

```
S.E.I.
Univ. of Waterloo
Univ. of Victoria
WCRE & ASE
```

The program describes a trip starting in Amsterdam on a certain date. It shows that we will fly via Atlanta to Pittsburgh to work at the SEI, then some time later we travel from Pittsburgh to the University of Waterloo via Toronto. Next, we fly to the University of Victoria via Toronto and Vancouver, then to Honolulu via Vancouver to visit two conferences. Some time later, we fly to Amsterdam via New York. Note the dead code: we do not fly through Detroit. Complex vouchers often have some phantom destinations simply because a two-way ticket is often cheaper than a one-way ticket. The indi-

Figure 4. Example Cobol code: (a) the original code; (b) heavily restructured code.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TRAVEL.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 D PIC 9(6) VALUE 980912.
01 X PIC 9 VALUE 1.
PROCEDURE DIVISION.
TRAVEL SECTION.
AMSTERDAM.
  IF D = 980912
    GO ATLANTA.
  GO HOME.
LOS-ANGELES.
  GO NEW-YORK.
HONOLULU.
  DISPLAY 'WCRE & ASE'
  ADD 14 TO D
  GO LOS-ANGELES.
DETROIT.
  DISPLAY 'NOBODY'.
WATERLOO.
  DISPLAY 'Univ. of Waterloo'
  ADD 6 TO D
  MOVE 0 TO X
  GO TORONTO.
ATLANTA.
  GO PITTSBURGH.
NEW-YORK.
  GO AMSTERDAM.
VANCOUVER.
  IF X = 0
    GO VICTORIA.
  GO HONOLULU.
PITTSBURGH.
  DISPLAY 'S.E.I.'
  ADD 14 TO D
  GO TORONTO.
VICTORIA.
  DISPLAY 'Univ. of Victoria'
  ADD 4 TO D
  MOVE 1 TO X
  GO VANCOUVER.
TORONTO.
  IF X = 1
    GO WATERLOO.
  GO VANCOUVER.
HOME.
  STOP RUN.
```

(a)

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TRAVEL.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 D PIC 9(6) VALUE 980912.
01 X PIC 9 VALUE 1.
PROCEDURE DIVISION.
TRAVEL SECTION.
AMSTERDAM.
  PERFORM TEST BEFORE UNTIL
       (D <> 980912)
    PERFORM PITTSBURGH
    PERFORM TORONTO
  END-PERFORM
  STOP RUN.
BAR SECTION.
BAR-PARAGRAPH.
  STOP RUN.
TRAVEL-SUBROUTINES SECTION.
PITTSBURGH.
  DISPLAY 'S.E.I.'
  ADD 14 TO D.
VICTORIA.
  DISPLAY 'Univ. of Victoria'
  ADD 4 TO D
  MOVE 1 TO X.
WATERLOO.
  DISPLAY 'Univ. of Waterloo'
  ADD 6 TO D
  MOVE 0 TO X.
TORONTO.
  PERFORM TEST BEFORE UNTIL
       X <> 1
    PERFORM WATERLOO
  END-PERFORM
  PERFORM TEST BEFORE UNTIL
       X <> 0
  PERFORM VICTORIA
END-PERFORM
DISPLAY 'WCRE & ASE'
ADD 14 TO D.
```

(b)

rect code (represented by flight transfers in the code) and the dead code (or phantom destinations) are typical for legacy systems. Also note the typical use of jump instructions that degenerate a program over time. With some effort, it is also possible to figure out that the restructured program's semantics are the same as the original program's. Note that dead code, indirect code, and GO TOs are gone, a simulated subroutine section has been created, subroutine candidates are present, and so on. So a lot of work has been done to simulate C code and shape up the Cobol. Now we are ready to perform

the syntax swap. As you can see, it is not the hardest part of the conversion. The result is

```
#include <stdio.h>
long D = 980912 ;
int X = 1 ;
void PITTSBURGH ( ) {
  printf("S.E.I.\n");
  D += 14;
}
void VICTORIA ( ) {
  printf("Univ. of
    Victoria\n");
  D += 4;
  X = 1;
}
void WATERLOO ( ) {
  printf("Univ. of
    Waterloo\n");
  D += 6;
  X = 0;
}
void TORONTO ( ) {
  while ( X == 1 ) {
    WATERLOO ( ) ;
  };
  while ( X == 0 ) {
    VICTORIA ( ) ;
  };
  printf("WCRE & ASE\n");
  D += 14;
}
void main ( ) {
  while ( D == 980912 ) {
    PITTSBURGH ( ) ;
    TORONTO ( ) ;
  };
  exit ( ) ;
}
```

Further restructuring of the C code is necessary. This dialect of Cobol had no native support for functions, but C does. We can collapse near-clones in the new C code into functions using parameters. A second step is to turn the global variables into local ones. Yet another step is to turn indirect code, such as a function call that only calls another function, into direct code. After performing these typical C restructuring steps, we end up with

```
#include <stdio.h>
void f(long dD, int
newX,long *D,int *X, char *s) {
        printf(s);
        *D += dD;
        *X = newX;
}
void TORONTO (long *D, int *X ) {
  while ( *X == 1 ) {
        f(6,0,D,X,"Univ. of
          Waterloo\n");
  };
  while ( *X == 0 ) {
        f(4,1,D,X,"Univ. of
          Victoria\n");
  };
  f(14,*X,D,X,"WCRE & ASE\n");
}
void main ( ) {
        long D = 980912;
        int X =1;
  while ( D == 980912 ) {
        f(14,X,&D,&X,
            "S.E.I.\n");
    TORONTO (&D,&X ) ;
  };
  exit ( ) ;
}
```

We obtained some C code, but this does not mean that language conversions are possible if you have the proper tools. The sample code is not the real thing: it does not have input, its output is trivial, there is no real use of data types, there are no possibly dangerous calculations, the output behavior is trivial, the program is small, the print routines are trivial, and so on. It merely illustrates the process of language conversion. Conversion technology (for instance, systolic structuring algorithms[11]) is still in its infancy and must be developed to separate coordination from computation so that subroutines are revealed.

Conversion technology is still in its infancy and must be developed to separate coordination from computation so that subroutines are revealed.

The realities of language and dialect conversion projects can be summarized in five rules of thumb:

- Conversions are difficult.
- Conversions are always more difficult than you think.
- The more semantic equivalence is necessary, the more impossible it gets.
- Going from a rich language to a minimal language is impossible.
- Easy conversion is an oxymoron.

Thus, we hope that more people will accept the realities of language conversions and that decision makers will limit their expectations regarding both the quality and the semantic equivalence of converted code. We also hope that software developers will use our and their own examples as an antidote to the technological quackery of language conversion vendors.

Much work must be done before language converters give satisfactory results on real-world code. As soon as we realize that language conversions are as easy as turning a sausage into a pig, our mind-sets are ready to attack the problem and considerable progress becomes possible.

Acknowledgments

Thanks to Alex Sellink (Quack.com) for writing the funny but instructive example program summarizing the travel scheme of a trip he and Chris Verhoef once made. We thank Edmund Arranga (Object-Z Systems), Eggie van Buiten (ASB), Bob Diertens (University of Amsterdam), Hayco de Jong and Jurgen Vinju (both CWI), Tim Bickmore (MIT), Robert Filman (NASA), Tom Holmes and Jasper Kamperman (Reasoning Inc.), Carl Gehr (Edge Information Group), Kostas Kontogiannis (University of Waterloo), Steve McConnell (Construx Software), Boris Kazansky and Mikhail Popov (Lanit-Tercom), Karina Terekhova (Oxford University), Gert Veltink (RogueWave Software), and the reviewers for their valuable comments and help.

About the Authors

Andrey A. Terekhov is a PhD candidate in software engineering at St. Petersburg State University. He is also working at Lanit-Tercom Company on contract with Relativity Technologies, Inc. (USA) on the creation of a software reengineering workbench, where he is a project manager. His research currently focuses on compiler technology, software reengineering, and cryptography. He received his MSc with honor from the Faculty of Mathematics and Mechanics of St. Petersburg State University. He is a member of the ACM and the IEEE Computer Society. Contact him at St. Petersburg State University, Faculty of Mathematics and Mechanics, 198504, Bibliotechnaya pl. 2, St. Petersburg, Russia; ddt@tepkom.ru; http://users.tepkom.ru/ddt.

Chris Verhoef is a computer science professor at the Free University of Amsterdam and is affiliated with the Software Engineering Institute of the Carnegie Mellon University. His research interests are software engineering, maintenance, renovation, software architecture, and theoretical computer science. He has consulted for hardware companies, telecommunications companies, financial enterprises, software renovation companies, and large service providers. He is an elected Executive Board member and vice-chair, conferences, of the IEEE Computer Society Technical Council on Software Engineering. Contact him at the Free University of Amsterdam, De Boelelaan 1081-A, 1081 HV Amsterdam, The Netherlands; x@cs.vu.nl; www.cs.vu.nl/~x.

References

1. T. DeMarco and T. Lister, *Peopleware—Productive Projects and Teams*, Dorset House, New York, 1987, p. 30.
2. C. Jones, *Assessment and Control of Software Risks*, Prentice Hall, Englewood Cliffs, N.J., 1994.
3. R.L. Glass, *Computing Calamities—Lessons Learned from Products, Projects, and Companies That Failed*, Prentice Hall, Englewood Cliffs, N.J., 1999, pp. 190–191.
4. R. Gray, T. Bickmore, and S. Williams, "Reengineering Cobol Systems to Ada," *Proc. Seventh Annual Air Force/Army/Navy Software Technology Conf.*, US Dept. of Defense, Hill Air Force Base, Ogden, Utah, 1995.
5. K. Kontogiannis et al., "Code Migration through Transformations: An Experience Report," *Proc. IBM Center for Advanced Studies Conf. (CASCON '98)*, IBM, Armonk, N.Y., 1998, pp. 1–12; www.swen.uwaterloo.ca/~kostas/migration98.ps (current Nov. 2000).
6. W. Polak, L.D. Nelson, and T.W. Bickmore, "Reengineering IMS Databases to Relational Systems," *Proc. Seventh Annual Air Force/Army/Navy Software Technology Conf.*, US Dept. of Defense, Hill Air Force Base, Ogden, Utah, 1995.
7. K. Yasumatsu and N. Doi, "SPiCE: A System for Translating Smalltalk Programs Into a C Environment," *IEEE Trans. Software Eng.*, Vol. 21, No. 11, 1995, pp. 902–912.
8. R.C. Waters, "Program Translation via Abstraction and Reimplementation," *IEEE Trans. Software Eng.*, Vol. 14, No. 8, 1988, pp. 1207–1228.
9. F.W. Calliss, "Problems with Automatic Restructurers," *ACM SIGPLAN Notices*, Vol. 23, No. 3, Mar. 1988, pp. 13–23.
10. J.C. Miller and B.M. Strauss, "Implications of Automated Restructuring of COBOL," *ACM SIGPLAN Notices*, Vol. 22, No. 6, June 1987, pp. 76–82.
11. M.P.A. Sellink, H.M. Sneed, and C. Verhoef, "Restructuring of COBOL/CICS Legacy Systems," *Proc. Third European Conf. Maintenance and Reengineering*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1999, pp. 72–82; www.cs.vu.nl/~x/cics/cics.html (current Nov. 2000).
12. H.M. Sneed, *Objektorientierte Softwaremigration [Object-Oriented Software Migration]*, Addison Wesley Longman, Bonn, Germany, 1998.
13. I. Jacobson and F. Lindström, "Re-Engineering of Old Systems to an Object-Oriented Architecture," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*, ACM, New York, 1991, pp. 340–350.
14. S. Oualline, *Practical C Programming*, 3rd ed., O'Reilly & Assoc., Cambridge, Mass., 1997.
15. W.M. Klein, *OldBOL to NewBOL: A COBOL Migration Tutorial for IBM*, Merant Publishing, Rockville, Md., 1998.
16. Y. Chae and S. Rogers, *Successful COBOL Upgrades: Highlights and Programming Techniques*, John Wiley and Sons, New York, 1999.
17. R. Widmer, *COBOL Migration Planning*, Edge Information Group, Mt. Prospect, Ill., 1998.
18. M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef, "Generation of Components for Software Renovation Factories from Context-Free Grammars," *Science of Computer Programming*, Vol. 36, No. 2–3, Mar. 2000, pp. 209–266; www.cs.vu.nl/~x/scp/scp. html (current Nov. 2000).
19. *COBOL/370 Migration Guide, release 1*, IBM Corp., Armonk., N.Y., 1992.
20. *VS COBOL II. Application Programming Language Reference*, 4th ed., IBM Corp., Armonk., N.Y., 1993.
21. C. Cerf and V. Navasky, *The Experts Speak—The Definitive Compendium of Authoritative Misinformation*, Villard Books, New York, 1998.