

The Realities of Language Conversions

A.A. Terekhov* and C. Verhoef**

**St. Petersburg State University, Faculty of Mathematics and Mechanics,
and LANIT-TERCOM
198904, Bibliotechnaya pl. 2, Peterhof, St. Petersburg, Russia*

***University of Amsterdam, Programming Research Group,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*

`ddt@tepkom.ru, x@wins.uva.nl`

Abstract

Billions of lines that have been written in COBOL, PL/I and other old programming languages are still in active use. Many commercial efforts to convert these languages to more modern languages have been started. However, few successful results have been achieved so far, which makes one wonder about the reasons why. This paper aims to shed light on the realities of language conversions and discusses some of the possibilities and limitations of automated language converters. We argue that the difficulties of source-to-source conversion are manifold and grossly underestimated. We illustrate that it is very challenging to resolve the intricate problems connected to language conversions.

Categories and Subject Description: D.2.6 [**Software Engineering**]: Programming Environments—Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring; D.3.4. [**Processors**]: Parsing.

Additional Key Words and Phrases: Reengineering, System renovation, Software renovation factories, Language conversion, COBOL, Visual Basic, Java, Pascal.

1 Introduction

It is well-known that the most influential cost drivers of software engineering are related to management, personnel and team capability. At the same time the economic impact of the use of software tools is relatively small; still this category is emphasized both by software vendors and by academic researchers. Possibly due to the inherent complexity of software systems, lots of managers are desperate enough to become victims of some technical quack that promises them significant improvements in software modification productivity. This mechanism, also known as the silver bullet syndrome, is what anthropologists call *name magic*. To apply name magic, you just say the name of the thing—COBOL to Java—and you have its full power at your disposal. Name magic, as the term indicates, does not need any proof supporting the claims. The claims are wholeheartedly accepted by anyone whose need for solutions is sufficiently large [5, p. 30].

Usually, large existing software portfolios severely limit the ease of making modifications. Due to earlier modifications the software deteriorates and it will

not come as a surprise that some of us draw the conclusion that reengineering software is a considerably harder problem than the task of green field software development. As the authors of [19, p. 7] observe, reengineering can be viewed as constrained problem solving because:

with reengineering, a solution—the legacy system—already exists and must be considered when developing (evolving) a solution to the new problem. The legacy system imposes certain restrictions on the problem solving activity that might not otherwise exist in a completely new engineering effort.

Many decision makers in the software industry find themselves trapped in a gazillion lines of legacy code, in dire need for 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. Jones [11] 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 with a language converter your personnel problems could be solved: a language conversion is supposed to bridge the gap between available knowledge of people and the knowledge that is necessary to solve legacy problems.

How hard is a language conversion anyway? 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, an academic researcher formulated 10 challenges for the next century. One of them was language conversions. Not just simple syntactic conversions, but conversions with a paradigm shift. From such observations one could draw the conclusion that automated language conversions are easy. And let's face it, converting the COBOL calculation `ADD 1.005 TO A` to its equivalent in Visual Basic `A = A + 1.005` is indeed very easy. Any simple tool can do it.

On the other hand, it seems that language conversions are a risky business. We know of three companies that went bankrupt and two departments of large software intensive enterprises that were dismantled. The reason for their untimely demise was in all cases 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 paper told us of a project where 50 million USD was spent on failed language conversion projects. Speaking of failure, Robert Glass—who collects failures—also illustrates our point in his book on computing calamities. He mentioned the failure of a translation system that would convert software from an obsolete system to a new one. Management told them that the conversion problem was limited in scope, and that only a limited set of constructs was used. This, turned out not to be true. The post-mortem analysis indicated that the converter was perhaps 10 times as complicated as was expected, and suddenly what had been technically feasible was becoming economically and technically infeasible [6, p. 190-1]. 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 that there is a lot of disagreement on the complexity of language conversions. This paper is devoted to the realities of language conversion tools. We focus on the issues that play a role when converting source code to other languages. We will argue that the problem of automated language conversions is grossly underestimated. We try to clarify the cause of the misconceptions with respect to the ease of language conversions. More importantly, we provide simple examples that expose the problems—anyone involved in software engineering can understand them.

Maybe it is theoretically possible to automatically improve the structure of a software system to achieve a paradigm shift, like introducing object-oriented concepts. However, this is very difficult to automate, and when large amounts of software have to be converted, this implies a lot of human intervention. In [7] it is stated that for paradigm shift translations extensive manual reworking is necessary. However, the payoff, especially for indefinitely-lived systems, is commensurately large. Due to the automation problems, most conversion tools apply the technology of syntactic conversion. Even with this perceived to be simple and low-level approach, many difficulties occur. We estimate that the scale of those intricacies is not yet fully understood. We hope that advocates of language conversions will limit their expectations on both the quality and the semantical equivalence of the converted code after reading this paper. We also hope that readers learn from this paper that the problems are deep and that they use our and their own examples as an antidote to the technological quackery of language conversion vendors.

There are not many papers that deal with the subject of this paper. The following papers are useful to study [13, 7, 16, 22, 20, 2, 14, 17, 18], but note that the most useful references are difficult to find or are in German. Contact the authors for copies or pointers.

Despite the lack of solid publications on language conversions there are many vendors advertising for language migration tools and services. Searching the Internet shows an abundancy of companies who can convert your systems to whatever other language. Although we are certain that some of these companies are doing a great job, there are also companies who claim to have technology and skills that happen to be insufficient. For instance, one of the companies that advertised on the Internet provided examples of converted code that were not even compilable.

Or there was a company claiming to be able to convert PowerBuilder to Java, but after our inquiries they did neither have experience nor tools to do the job. Instead they were claiming to have a process! Sneed [18] summarizes the state-of-practice of the transformation marketplace in his book on object-oriented software migrations as follows:

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.¹

¹Warning: this quote was converted from German to English by the authors of this paper.

Organization The rest of this paper is organized as follows. In Section 2 we discuss some typical requirements for language conversion tools. Section 3 describes in detail problems and inconsistencies associated with the task of translation from one language to another. In Section 4 we provide a process for language conversions, and we illustrate it with a very simple example. Finally, in Section 5 we draw some conclusions.

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

2 Requirements for Conversion Tools

The problem statement for any language or dialect conversion is simple: convert this system to that language or dialect while the external behavior does not change. From an abstract point of view, a language migration project seems deceptively simple. As a consequence, it is our experience that requirements for language converters are often not properly formulated. In this section we will elaborate on the requirements for language converters.

Most of the times the ease to formulate a solution for a particular problem is caused by the availability of constructions that facilitate expressing the solution conveniently. 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 have no choice but to use the latter language, we have to simulate the conditional construct. Such code fragments could be called *simulated language constructs*. We can, for instance, simulate object-orientation in a language lacking OO-support.

The task to convert a software system into a new language amounts to a mapping of the language constructs used in the system to new language constructs in the new language. We present a picture of this mapping in Figure 1. There are at least six possible categories in this mapping. During language conversions the authors have encountered examples of all these categories. In requirements specifications for language converters, usually only the native-to-native part of the mapping is mentioned in the form of a statement-by-statement conversion. This phenomenon is in accordance with the tendency of people to first focus on the easiest problems. The second author has been external reviewer of several large-scale language conversion projects. Most of them ended in failure because the conversion problems were not addressed in the requirements. For instance, in one case about 80% of the requirements specification was devoted to the graphical user interface, while the actual language converter

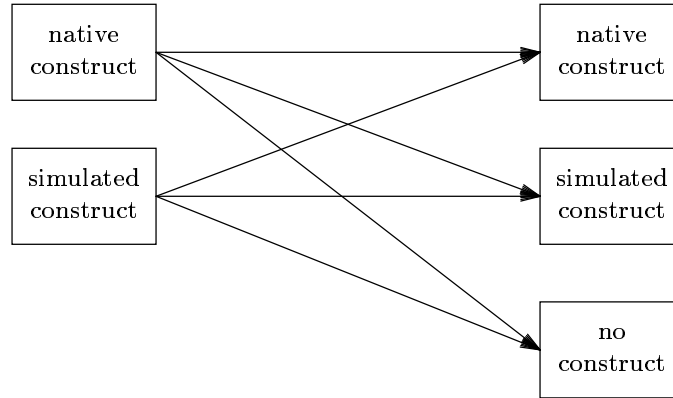


Figure 1: A mapping of constructions between languages.

was represented by a single arrow, as if that part was trivial. Because the problem was underestimated, the real language conversion issues were omitted in the requirements. This often leads to runaway projects. Namely, first-time failure causes the actual conversion of the software to be delayed. This, in turn leads to increased pressure on a new development team to deliver the converter. The large pressure makes it tempting for the new team to skip requirements all together, which closes the positive feedback loop. After a few failure cycles, total break down exits the loop. This positive feedback loop caused the demise of the earlier mentioned language conversion companies and departments.

We list a few requirements that we think are necessary for realistic development of source-to-source converters.

- We need an inventory of the native and simulated language constructs of the system that needs conversion.
- For each language construct a conversion strategy should be developed. More specifically, a list of input and output fragments should show what the desired behavior of the converter is. Recall the failure reported on in Glass's book: apparently no such fragments were collected from the original system.
- It shall be stated whether the converted system should be functionally equivalent to the original system. We note that the first intuition for this requirement is always that both systems should be equivalent. However, in practice, a sophisticated automated modification effort usually exposes faults and unsafe code in the original system. Often, the customer then requires that during automated modification, the faults are taken care of as well. This requirements creep should be taken care of in the requirements. Note that it also complicates testing the new system because a regression test is based on equivalence.
- It should be stated whether the test sets belonging to the original system also have to be converted. During such automated efforts, also errors in the tests are exposed. It should be clearly stated what is the policy towards modification of the test sets.

- We shall strive for maximal automation of the conversion, that is, reducing the human interference by a conversion team.
- If the converted system becomes the primary source code of the system, this converted system must be maintainable. If for instance the maintenance team of the original system is going to continue maintenance, an often heard requirement is that the converted system should be as similar as possible to the original system, so that the maintenance team can recognize the original code. If on the other hand, the maintenance team is new, then the conversion should try to use the idiom that is common in the target language so that maintainers recognize the code as normal for that language.
- The converted system should be of acceptable efficiency both in compilation and execution time.
- If the converter is needed many times instead of once, also the conversion time required for conversion itself is a relevant issue. We note up-front that it is not always trivial to optimize this without distributing calculations over many machines.
- The size of the converted system must not considerably exceed the size of the original system if it is to be maintained.

Note that, apart from these explicit requirements, there is always an implicit understanding of how the converter will work. It reflects the user's expectations of the advantages associated with transferring the system to a more contemporary environment. Very often these imaginary advantages motivate the process of conversion. In most cases, however, such expectations never realize in full, which causes disappointment. A popular misconception is that after conversion the system is change-enabled so that totally new features can easily be added. The problem is aggravated by companies marketing conversion software as yet another silver bullet. The quality of such converters is often not optimal and in some cases nonexistent.

We recall from the introduction the company who stated: "The converter runs as a simple wizard: it is intended to be something a secretary can run." We believe that with a well-designed interface nonprofessional programmers indeed can use almost any system, including a COBOL to Visual Basic converter. We will argue in Section 3 that the problems of converting such code asks for highly skilled programmers who know about the intricacies of both the original and the target language. So it is senseless to let a secretary run a conversion tool.

Another issue is that often an automatically converted program is never as good as a new one developed within the full range of a contemporary programming language. Although this criterion is often quoted as the ideal (see [20]), in practice the converted programs often retain the idiom of the source language (see Section 3.3 for an example). It may be fair to expect that the structure of the program will only improve after conversion, but conceptual changes to the application will always remain labour-intensive and require human interference (as also described in [10, 7]).

3 Technical problems with Language Conversions

In this section we will illustrate the requirement that a list of input and output patterns is very helpful when converting between languages or dialects. Such examples establish the core of language conversions. They also give insight in the realities of language conversions for anyone who is considering to migrate to a new language or dialect.

3.1 Conversion of Data Types

One of the first problems that we have to deal with when building a converter is the conversion of data types. Although we do not always realize it, programming languages usually have idiosyncratic data type conventions. To explain what we mean let us take a look at C++ and Java. Many people consider them to be very similar, so a native-to-native conversion seems to be a simple task. Still the data types reveal many differences. For example, there are no pointer type variables in Java that are present in C++. On the other hand, there are no Booleans in C++ but they are present in Java. Also the data type sizes vary from platform to platform for C++, whereas they are fixed in Java. So even while converting between the languages C++ and Java, we run immediately into the problem of representing idiosyncratic data types.

When we take two perceived-to-be-similar languages and detect so many differences that have to be taken care of, it should not amaze anyone that the differences between languages like COBOL or PL/I and languages such as Java, Visual Basic, or C++ are perhaps unsurmountable. For example, consider the PL/I data type below.

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

The variable C occupies 3 bytes with decimal point assumed 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, this data type and also the assignment operator do not correspond to any of the standard C++ data types and assignment operations².

3.2 COBOL to Visual Basic

We discuss the issue of data types a bit more while focusing on a COBOL to Visual Basic conversion example. One possibility is to convert only those data types that have an equivalent in the target language. So the native-to-native part of the mapping in Figure 1. Variables of all other data types are reported to the user with the advise to rewrite parts of the code that use them. Let us have a look at a simple COBOL code fragment to illustrate the problems with data type conversions.

²Note that the paper [13] converts a PL/I dialect to C++ but does not address this problem.

```

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

```

The variable `A` is able to represent values like -3.1416 . Initially the contents of `A` represents the number -1 . In the procedural part of the COBOL program, we add the number `1.005` to `A`. Finally, we print the result to the screen. A converter could turn this simple COBOL program into the following Visual Basic program:

```

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

```

The Visual Basic Program declares the same variable `A` as a `Double`. The variable `A` is initialized and obtains the value -1 . The COBOL `ADD` is represented as an addition in the Visual Basic program. When we run both programs, the Visual Basic code prints a different result than the COBOL code. The visual Basic code yields $+0.0049$, indicating a rounding error.

The reason for the rounding error in our first Visual Basic conversion attempt is due to the fact that the COBOL code uses a fixed-point data type and the Visual Basic fragment uses a floating-point variable. The poor precision of the Visual Basic floating-point implementation has been documented in various Visual Basic reference books.

For the above program we can solve the rounding problem by using another data type, namely the Visual Basic `Currency` data type. This solves the rounding problem in the first Visual Basic program. 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 Visual Basic code prints 0.0001 , which is twice as much as in the initial COBOL program. This error occurs since the `Currency` data type uses 4 digits precision and uses rounded results for smaller amounts. So the `Currency` data type will misbehave in other contexts.

So, there is no single data type in Visual Basic that can handle the fixed-length record structure in COBOL. Therefore, *any* simple strategy to convert

these COBOL data types is doomed to fail, because in different contexts different solutions are necessary. This makes clear that a sophisticated data type analysis is mandatory even in the above simple example programs.

3.3 COBOL to C

We mailed the academic researcher mentioned in the introduction the first simple COBOL program and its conversion to Visual Basic to check whether he considered this to be a syntactic transformation. He confirmed this and stated about this type of transformations:

I contrast syntactic transformation (which we know how to do well),
with deeper ones.

First of all, he did not observe the rounding problem in the Visual Basic conversion. This is perfectly understandable, since he told us later that he never worked with COBOL nor Visual Basic, nor on a converter between the two. Secondly, he spontaneously converted our first COBOL example into C to illustrate his point:

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

This code is not compilable 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 expect floating point 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 [15]). 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. Some 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. On the positive side, we do have the right arithmetic answer as is also obtained in the COBOL program, on the negative side, the output of the COBOL program is different from the

output on the C program. We did not mention this before, since we want to address one problem at a time. The actual output of the COBOL program is: 00050+. The literal output of the above C program is 0.005000. Obviously, the C program is not semantically equivalent to the COBOL program. This may seem nit-picking, but let us suppose that the above COBOL code is part of a very successful multi-lingual invoice layout system where printing in the wrong format has devastating consequences. After all, only business critical systems are candidates for conversion efforts. In a next attempt to convert the COBOL program we come up with the following code.

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

Observe that this program has little resemblance to the initial solution. Secondly, 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 preserved in the C code.

Let's discuss the code. We see the initial stage of a library function `display` in the code that emulates the output behavior of the COBOL `DISPLAY` statement. We declare a character string of length 16 and we 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 to 0, which is the exit status of our C program? More importantly, we hope that the reader will be in doubt whether our C program is indeed equivalent with the COBOL program. For, this is a healthy viewpoint when it comes to converted code.

3.4 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; we call such problems *data type abuse*. As an example, 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 variable `A` that is a short integer 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, while the Java program displays 100.

Solving the problems of unexpected side-effects with data types is one of the realities of language conversions. Fighting them can be done using a different approach than the one we discussed thus far. For each source language data type which does not have a *precise* equivalent in the target language, dedicated support emulating the transactions specific to this data type is created. We call this *data type emulation*. We note that in [22] an example of data type emulation for converting Smalltalk classes into C is present. So in a sense we add to the target language some constructs 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. One could call such programs, Java-compliant COBOL programs, but maybe not Java programs. As soon as we start to emulate the 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? In many cases 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

we can overload the assignment operator =. 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 have to deal with, it becomes really 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.

3.5 OS/VIS COBOL to VIS COBOL II

We have seen that it is not so easy to convert the data part of programs in a safe manner to the target language. Also the conversion of the procedural code is not an easy task. So let us limit our expectations a little and look at the subject of dialect conversions. In this section we will focus on converting between various COBOL dialects for IBM Mainframes. The perception of many of us is that such conversions are a non issue. Let's have a look and find out. The following code excerpt displays the word IEEE:

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

In all dialects of COBOL this is valid syntax. So it is tempting to assume that there is no conversion necessary. The problem that we have with dialect conversions, in addition to language conversions, is that the *same* syntax can have *different* behavior. For instance, the OV/VIS COBOL compiler prints the expected result, namely ' IEEE', which is right justified. The COBOL/370 compiler displays the output 'IEEE ' with a trailing space, which is *left* justified³. This is not an isolated case. There are many more problems of what Carl Gehr and Rex Widmer call "same syntax, different behavior" in their 500 page COBOL Migration Course [21]. We call this the *homonym problem*. Also the books [12, 4] could be studied.

We also have detectable compatibility problems. The arrow from native-to-simulated in Figure 1 comprises the conversion of language-specific constructs in the source language that do not have simple equivalents in the target one. There is an incompatibility in the date routine of various dialects. Consider the following COBOL programs taken from [1].

<pre>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.</pre>	<pre>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.</pre>
---	---

³If you wonder why, we have two words for you: ANSI Standards

```

DISPLAY 'MONTH = ' H-MM.    DISPLAY 'MONTH = ' H-MM.
DISPLAY 'YEAR  = ' H-YY.    DISPLAY 'YEAR  = ' H-YY.

```

Basically, the left-hand side program declares a TMP variable with eight positions. A data type H-DATE is defined for dealing with dates like 13/01/99. In the procedural part, the special register CURRENT-DATE stores the date of today in the TMP variable. This value is stored in the simulated date field. Finally, the day, month and year are printed. When we wish to convert this program to a newer dialect of COBOL we have to 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 being the type of the CURRENT-DATE. In the COBOL/370 Migration Guide of IBM [8] it is proposed to convert the type of variable TMP from PIC X(8) to X(6) and to convert MOVE CURRENT-DATE into an ACCEPT statement. This leads to the right-hand side program.

The IBM solution breaks down in the above context, since 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 in the converted system. So the output of the converted program is completely erroneous. We ran the converted program on September 15, 1999. The converted program shows this output:

```

DAY    = 91
MONTH  = 99
YEAR   =

```

What happens is that the string 990915 is mapped on the data item H-DATE. Subfield H-MM obtains the first two digits and is set to 99, then the FILLER is set to 0. The H-DD gets the next two digits: 91, the next FILLER gobbles up the last 5, and H-YY gets nothing. Then the program prints first the contents of H-DD with value 91, then H-MM leads to 99, and the H-YY prints nothing.

How to solve this? One solution could be to also convert the H-DATE field, and all code that depends on the H-DATE field. This solution ripples through the program. Moreover, it can affect other programs as well: the data type could be used in a data base or passed as parameter in a call to another program. The conversion of the procedural code is tightly coupled with the conversion of the data types involved in the procedural code. If we follow the IBM solution, the entire system needs to be modified. When we use data type emulation, we can prevent the ripple effect satisfactory, namely by converting to the following code [1]:

```

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 new system call that we have to use in the procedural code. In the procedural code 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 the TMP variable. Now all the other code that is relying on the old date type runs as if the old special register is used. The entire ripple effect is prevented, and the solution can be automated for 100%. It may be not as beautiful as it could be. We note that IBM has corrected the abovementioned error in newer versions of their migration guide.

3.6 Turbo Pascal to Java

Let us abstract for a while from the data type problems and look at the following Turbo Pascal program. The 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.

```

There are 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 we replace in the string abc on the first occurrence of substring a, the string d and we write the string to the screen. So the output of this program is 0 and dbc. The following Java program could be the output of a 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);
}
}

```

In this code we declare `s` to be `abc` and we set `a` to the offset of the first occurrence of the string `d` in the string `s`. However, the convention in Java is that when there is no such occurrence 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);
    }
}

```

The solution is that we add 1 to the index to simulate the Turbo Pascal return-code. This solves the return-code problem in the first part. The value of `a` is now correct, namely 0. However, now the second part is incorrect: the output is `adc` instead of `dbc`. Apparently the second position in the string `abc` is replaced by a `d`. This is due to another side-effect of conversion: arrays in Java start with zero instead of 1 such as in Turbo Pascal. We also take this effect into account leading to the following output:

```

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 (viz. Figure 2) of the converted code we can normalize for cumulative calculations in the code and restructure the code below

```

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

```

by rewriting it into:

```

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

```

Now we understand why in the first conversion attempt, the second code snippet 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 the *Dilbert Principle*: Two wrongs make a right, almost.

3.7 The Gory Details

The essence of reengineering is getting all the gory details right. In this section we indicate a few of these details to make sure that the reader will understand that the list of annoying issues that need to 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 based on code from an IBM manual [9]):

```
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. Moreover, when there are too many elements 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 has to be emulated when converting into a language not supporting this kind of composition of variables. In order to preserve semantical correctness of the program, 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 former depends on a solution to the latter. 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 with this kind of operations in them. 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 counted (tallied) or replaced or both. Automated conversion to languages not supporting this kind of functionality in a native way, need to be extended with syntax and semantics capturing such constructs, which brings us to the more general issue of the domain focus for a language. The above example 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, a language with a different domain focus 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 such 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 in general is *always* intricate.

3.8 Discussion

The examples that we presented clearly illustrate that language conversions are grossly underestimated—even by well-known experts in the area of reverse engineering. More information on experts and misinformation can be found in [3]. Even when we restrict ourselves to a dialect conversion of two IBM products, we encounter problems. It seems that IBM underestimated the conversion of their own dialects.

The examples clearly illustrate that automated language conversions are much more difficult than many people anticipated. A possible cause of underestimating the problems is that the surface syntax of the converted arithmetic looks deceptively similar to the original. Both calculations `ADD 1.005 TO A` and `A = A + 1.005` *look* the same, but their implementations are two *distinct* approximations of elementary school arithmetic as popularized by Sesame Street. We are fooled by our perception of arithmetic reasoning. Moreover, the printing routines `DISPLAY`, `MsgBox`, `printf` also have the same look and feel, but their semantics is not in accordance with our intuitions. Finally, when we restrict to dialect conversions, the problem becomes even harder: there we are dealing with programs that compile under many compilers, but the semantics of the same syntax differs from compiler to compiler. As a consequence, the semantics of converted code will usually differ from the original unless many precautions are taken.

The examples also show that it is very dangerous to map data types of one language to an approximate data type in the target language. In fact, the examples that illustrate this are those parts in Figure 1 that map 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 native data types of the target language may simplify maintenance since it delivers less alien code, but this approach clearly reduces the level of automation or affects the semantical correctness of the result. Applying data type emulation leads to more automating and more correct programs, but at a price of extra work required for writing runtime libraries, less simple maintenance and loss of efficiency in the performance of the converted system.

Both techniques can be used simultaneously. It is for instance possible to first analyze a source program whether it is *type safe*. That is, we check that the 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 convert 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 for the complexity of a conversion project. 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. Freshmen would expect that the more equal the

languages are, the more easy a conversion would be. The homonym problem shows the opposite: in addition to all the problems we have with language conversions we now also have to deal with semantic differences that we cannot even detect syntactically.

Any decision maker considering language or dialect conversions to solve some problem, should realize that the problems that are perceived to be solved by the conversion will actually be replaced by other maybe 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.

4 A Process for Conversion

Converting application software from one language to another is usually done to simplify further maintenance. Therefore, the target source texts must be well-structured, contain as little global data as possible and so on. Since the source language programs are rarely compliant with these requirements, there is an urgent need for restructuring them before conversion. So any sensible language conversion should first start with extensive restructuring. We note that in 1988 [2] already a number of problems with classical restructuring tools were enumerated.

In this section we will show some important steps that are necessary when dealing with language conversions. The accompanying simple example will illustrate some aspects of them. The example is based on work that was done for a Swiss bank [17]. The basic process of language conversions as we experienced it, is depicted in Figure 2.

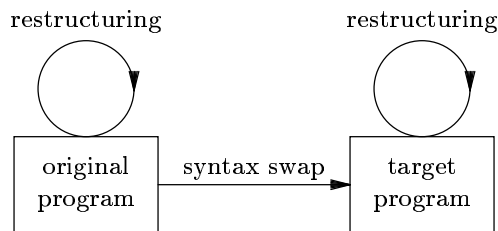


Figure 2: Process for language conversions.

First of all, the original program is restructured. This is not limited to restructuring in the sense of `G0 T0` elimination, but it is restructuring so that the unwanted arrows in Figure 1 are removed as much as possible. As an example, in COBOL there is not necessarily a `main` but in C it is common to have a `main`. Therefore, in a conversion from COBOL to C restructuring the COBOL code should be restructured such that a simulated `main` in the COBOL code ensues. Subroutines are not common in COBOL, but are in other languages. It is hard to recover them, especially with unstructured code. For example, in a COBOL to Ada conversion [7] the identification of subroutines was hampered by an abundance of `G0 T0`s in the COBOL code (every 225 lines of code a `G0 T0`).

When migrating to other languages usually restructuring steps will be pre-quisite to what we call the syntax swap. This is a relatively easy step where we swap the syntax of the precooked restructured original code to the target syntax. The swapped programs are usually very 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.

In order to illustrate the process, we discuss an example COBOL program that we are going to transform into C. The example COBOL code has been composed from actual legacy code from a Swiss bank. We abstracted from the data type emulation problems to focus on the problems with procedural code. The code is adapted to the knowledge domain of traveling so that the meaning of the program is easy to understand for all the readers. Note, however that the code is of the quality that you might expect for real conversions (every four lines a GO TO). At the left-hand side we see the original code, and the right-hand side is the heavily restructured COBOL code, using the technology studied in [17].

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

```

IF X = 1
  GO WATERLOO.
GO VANCOUVER.
HOME.
STOP RUN.

```

The output of both programs is as follows:

```

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

```

With a little effort it is possible to find out that this program describes a trip that starts in Amsterdam on a certain date. The program shows that we fly via Atlanta to Pittsburgh to visit the SEI. Then we fly some time later from Pittsburgh to the University of Waterloo via Toronto. Then we fly to the University of Victoria via Toronto and Vancouver. Then via Vancouver we fly to Honolulu to visit two conferences. Some time later we fly to Amsterdam via New York. Note the dead code: we never fly via Detroit. The indirect code (represented by flight transfers in the code) and the dead code 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 semantics of the restructured program is the same as the original program. Note that dead code is gone, indirect code is gone, 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 can be seen it is not the hardest part of the entire conversion. The result is as follows:

```

#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. For instance, in COBOL it is not customary to use functions with parameters. In C it is normal to use such notions. 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, like a function call that only calls another function, into direct code. After performing these typical C restructuring steps we end up with the following code:

```

#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 as you can see, it was not easy, even though we used proper tools. Also, this tiny program could not be regarded as a real-life example, for the program does not have input, its output is trivial, there is no real use of data types, there are no possibly dangerous calculations, the program is small, the print routines are trivial, and so on. It merely illustrates the process of language conversion. The example shows that extensive technology that is still in its infancy (for instance, systolic structuring algorithms [17]) needs to be developed in order to separate coordination from computation so that subroutines are revealed.

5 Conclusions

In this paper, we have explained why (automated) language conversion projects often fail. With a string of simple examples, all taken from real-world situations,

we have shown the true face of language conversions. Although the problem of automated source-to-source transformations looks deceptively simple, it is in fact a very complicated problem with many compromises. We summarize the realities of language and dialect conversion projects in the following 5 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

Finally, we hope that more people will accept the realities of language conversions, and that decision makers limit their expectations on both the quality and the semantical equivalence of conversion projects.

There is a lot of work to 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 it becomes possible to achieve considerable progress.

References

- [1] 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*, 36(2-3):209-266, March 2000.
- [2] F.W. Calliss. Problems with automatic restructurers. *ACM SIGPLAN Notices*, 23(3):13-23, March 1988.
- [3] C. Cerf and V. Navasky. *The Experts Speak - The Definitive Compendium of Authoritative Misinformation*. Villard Books, 1998.
- [4] Y. Chae and S. Rogers. *Successful COBOL Upgrades: Highlights and Programming Techniques*. John Wiley and Sons, 1999.
- [5] T. DeMarco and T. Lister. *Peopleware - Productive Projects and Teams*. Dorset House, 1987.
- [6] R.L. Glass. *Computing Calamities - Lessons learned from products, projects, and companies that failed*. Prentice Hall, 1999.
- [7] R. Gray, T. Bickmore, and S. Williams. Reengineering Cobol Systems to Ada. In *The Proceedings of the Seventh Annual Air Force/Army/Navy Software Technology Conference*, Salt Lake City, April 1995.
- [8] IBM Corporation. *COBOL/370 Migration Guide*, release 1 edition, 1992. Publication No. GC26-4764-01.
- [9] IBM Corporation. *VS COBOL II. Application Programming Language Reference*, 4 edition, 1993. Publication No. GC26-4047-07.
- [10] I. Jacobson and F. Lindström. Re-engineering of old systems to an object-oriented architecture. In A. Paepcke, editor, *OOPSLA '91: Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 340-350, 1991. Appeared as special issue: SIGPLAN notices 26(11):1-365.
- [11] C. Jones. *Assessment and Control of Software Risks*. Prentice-Hall, 1994.
- [12] W.M. Klein. *OldBOL to NewBOL: A COBOL Migration Tutorial for IBM*. Merant Publishing, 1998.

- [13] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Muller, and J. Mylopoulos. Code migration through transformations: An experience report. In *Proceedings of CASCON'98*, 1998.
- [14] J.C. Miller and B.M. Strauss. Implications of Automated restructuring of COBOL. *ACM SIGPLAN Notices*, 22(6):76–82, June 1987.
- [15] S. Oualline. *Practical C Programming*. O'Reilly & Associates, Inc., 3rd edition, 1997.
- [16] W. Polak, L.D. Nelson, and T.W. Bickmore. Reengineering IMS Databases to Relational Systems. In *The Proceedings of the Seventh Annual Air Force/Army/Navy Software Technology Conference*, Salt Lake City, April 1995.
- [17] M.P.A. Sellink, H.M. Sneed, and C. Verhoef. Restructuring of COBOL/CICS Legacy Systems. In P. Nesi and C. Verhoef, editors, *Proceedings of the Third European Conference on Maintenance and Reengineering*, pages 72–82. IEEE Computer Society Press, 1999.
- [18] H.M. Sneed. *Objektorientierte Softwaremigration*. Addison-Wesley, 1998. In German.
- [19] S.R. Tilley and D.B. Smith. Perspectives on legacy system reengineering, 1999. Available at <http://www.sei.cmu.edu/reengineering/pubs/lsysree/>.
- [20] R.C. Waters. Program translation via abstraction and reimplementaton. *IEEE Transactions on Software Engineering*, 14(8):1207–1228, 1988.
- [21] R. Widmer. *COBOL Migration Planning*. Edge Information Group, 1998.
- [22] K. Yasumatsu and N. Doi. SPiCE: A System for Translating Smalltalk Programs Into a C Environment. *IEEE Transactions on Software Engineering*, 21(11):902–912, 1995.