Re-animation of computer programs
Meijer, F.J.

Citation for published version (APA):
Meijer, F. J. (2001). Re-animation of computer programs

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
3 HISTORY OF THE PROBLEM

3.1 INTRODUCTION

In the foregoing chapter the main characteristics of the computer were discussed. An important characteristic of the development of computer technology is the rapid succession of changes. The differences between the former and current computers are significant. Not only do the machines differ in technical aspects, they also use different software. Usually innovation results in reduction in cost and increased performance. However, upgrading to newer computers poses a unique problem for users because low-level software of the old machines cannot be used on the new ones. As each type computer has its own low level language, software in this language has to be transformed to that of the new machine, otherwise it cannot run.

For data stored in the old computer the problem is less severe because they can be reformatted to make optimal use of the new hardware’s capabilities. The data storage problem has been alleviated by separating the physical storage (the way the data are stored) and logical storage (how the software uses it).

In this chapter a historical overview will be given of the problem that came to light when newer computers emerged that could not execute earlier machine code.

3.2 HISTORICAL DEVELOPMENT

The problem was soon recognized and solutions proposed. Manufacturers circumvented it by emulation. Users did not enthusiastically embrace this solution because it was too time consuming. The technical development proceeded unhampered by the opinion that it was harmful to the users, but beneficial to the producers. This continuing technical development compounded the problem. For example, new displays allowed for
emergence of different graphical user interfaces (GUI) that made it more difficult to exchange applications between different computers. (Andersen, 1991)

3.2.1 The problem: relatively old

The problem of how to run programs written for a specific computer on another is known in literature as the portability problem. It was considered important from the beginning of computer development. As Fairfield (1985) writes “Portability has been an aim of computer scientists virtually since the first computers were invented”. But, initially not much effort was invested to realize this aim. Hardware and software vendors were primarily concerned with technological advancements. For them solving the problem had a low priority. Users exposed to this problem complained. Their concerns fell on deaf ears. Devising appropriate software to transform old programs, in such a way that they could function on the new equipment, was initially considered as a solution. This solution was tried but not pursued. The problem moved to the background. Literature became concerned mainly with avoiding future problems. The proposals can be divided into two categories. The first entails developing hardware to run software developed for different platforms on the same computer. The other was to develop software with the same aim. These categories will be explored further in the following sections (see figure 2).
3.3 HARDWARE

If a computer has to run software written for an older machine, the hardware must not only understand the machine language developed for the new type computer, but also the language of the obsolete computer. Most CP/M machines used the Z80 processor. The IBM-PC and its compatibles succeeded it, using an 8086 processor. Computers were developed that could not only execute their own instructions but also those of another. Users could switch between instruction-sets. These computers were equipped with two different processors. However, they were not successful, they were expensive and, therefore, not widely used. Users that require different platforms prefer to buy separate systems. It does not appeal to users unfamiliar with both. Although it made the software of some platforms accessible, older ones could not be accessed. Furthermore, when upgrading, the new computer should incorporate the obsolete processors. For technical and economical reasons such computers have never been built. In some cases an additional piece of hardware can be purchased making it possible for the user to choose between two sets of instructions. This additional hardware contains a contemporary processor of another widely used computer. It is not developed for use with obsolete processors.

Other hardware was tried. Computers were introduced that were able to execute one higher-level language only. Initially, the personal computer market was saturated with equipment incorporating only the BASIC language. Part of the hardware consisted of a read only memory (ROM), containing software enabling the use of the language chosen for implementation. Programs could be directly executed without the need for compiling and linking. Interest was not sustained because further developments in personal computers made it possible to incorporate more sophisticated software. The companies that produced computers restricted to one higher-level language went bankrupt or discontinued production. Users considered the use of only one language too constraining.

1 Byte, vol. 6, no. 11, p. 505.
Byte, vol. 8, no. 6, pp. 150-164.

2 For example a computer executing instructions written in the language FORTH was produced.
3.4 SOFTWARE

Most efforts were sought in developing software, emulators, that would make it possible to run software devised for a specific computer on another one. Emulation lets the new computer behave as if it was the old one. In such a case the old software is emulated on the new computer and each instruction of the old computer has to be interpreted before execution. As a consequence a large variety of emulators are available. May (1987) identifies different generations of such emulators\(^3\). Using an emulator has some disadvantages. The most significant is the sluggishness of the emulation. This is due primarily to the proliferation of instructions required by the emulator. Until recently this offset the advantage of the faster object computer. Also the user is forced to work in an unfamiliar software environment. This is acceptable if the user wants to gain experience with the old environment. When IBM introduced a new model computer its software was not ready. However, it sold them with an emulator to run their existing software. The new IBM, with its emulator, reduced the speed of its programs to such an extent that the old outperformed the new.

Emulation makes it possible to run existing software. How to write programs that could run on different type computers was emphasized in early literature. The development of a universal language got special interest. All programming languages, including specific software applications, should be translatable to such a language\(^4\). Its reason for failure is explained in the next section.

3.4.1 UNCOL (Universal Computer Language)

Higher-level languages eased programming efforts on current machines\(^5\). Although different higher-level languages were supported they could not cope with differences in hardware. Selecting a specific language, and

---

\(^3\) May uses the term simulator. An emulator is a program that simulates a piece of computer hardware. Nowadays the notion 'simulator' is used generally for a program that simulates other systems. (e. g. weather forecasting programs, flight simulators)

\(^4\) Coleman (1974) proposed JANUS as a universal language. Also in 1975 Druseikis developed SiU2. This language was written for a hypothetical machine. Tanenbaum, in 1978, recommended IBM System/370 machine code as a universal 'between language'.

\(^5\) Higher languages are computer-programming languages that use natural language commands.
making it universal, seemed an attractive proposal. There was disagree­
ment over the selection of a universal problem oriented computer lan­
guage. Agreement was not achieved because the myriad of problems to be
solved required a variety of problem-oriented languages (Waite, 1977).
Instead of making one of the existing languages universal, the idea arose
to design a new one⁶. In the mid 50's the first suggestions for a universal
computer language emerged⁷. Coleman devised a universal intermediate
language called JANUS (Coleman, 1974). In 1958 Strong and others pro­
posed a universal computer-oriented language called UNCOL (Strong,
1958). They noted that every three to five years users acquired newer
type computers because of their superior technical capabilities. They also
noted that the cost involved in upgrading was high. Nevertheless, this did
not prevent such changes. The aim of the proposed UNCOL-system was
to translate programs, written in different problem oriented higher lan­
guages, into one language: UNCOL. Generators, as they were called, had
to be developed to perform the translation from programs written in a
specific problem oriented language to UNCOL. So, for each higher-level
language a generator would be needed. Next, UNCOL would translate
programs into machine language for the desired object computer. Each
type of machine would require generators and one translator (see figure
3). To cite a metaphor: this multistage rocket never got off the ground for
several reasons (Tanenbaum, 1983). The major one was the sheer audacity
of the approach. It implied considering all possible higher programming
languages and embedding them into only one. Even approximations to a
universal language of this nature were extremely difficult to achieve. Such
a language would be quite complex and too unwieldy to be able to be
implemented on computers available at that time. Furthermore, the
machine code produced by a compiler was far from optimal. This gener­
ally led to speed losses in the compiled program. If a program, written in
a higher-level language, should first be translated to a universal one, the
speed loss would be even greater.

⁶ In Jonathan Swift’s time (1749) attempts were made to create a universal language that would be
useful for the dissemination of scientific truths. (Swift, reprinted 1981, p 183)
⁷ According to Conway UNCOL was proposed as early as 1954.
3.4.2 The Program Transferability Study Group

The tone of the discussion on the problem how to run existing software on newer equipment gradually grew more pessimistic. It became apparent that it was difficult. Mc Carthy (1961) suggested calling it “a group exercise in wishful thinking”. Nevertheless, the idea was not totally set aside. In January 1968, the Program Transferability Study Group was established (Morenoff, 1969). In June of the same year the group announced their first findings and published them in December 1968 (Mealy, 1968). According to the group main obstacles were loose specification of data structures and lack of programming standardization. They argued that present languages provide too much freedom to the programmer. This would result in difficulty in transferring these programs. Although the use of higher-level languages eases the transfer of programs to other installations, they found their use too constraining. As a possible solution for existing software they proposed “decompiling” binary programs written for one system, and subsequently generating new code for another envi-

---

8 At that time there was no clear distinction between transferring and transforming software.
ronment. Another proposal was to set precise programming standards. The study group concluded that these proposals would not solve the problem for old software, but could possibly prevent future ones. Transformation of software in itself is difficult enough. It becomes more complicated if the software is poorly structured. Although they considered the use of high-level languages too constraining, they lingered with the idea of creating one universal language. However, languages focusing on one type of problem have a better chance of survival than a universal one. Insofar as the group concluded that existing higher level languages were inadequate because of the lack of important constraints, their efforts were not totally in vain.

3.4.3 Rules to avoid future problems

To avoid future problems the formulation of programmer rules was proposed. (Kaindl, 1988) This would alleviate the central problem of running software on future computers and also ease the task of implementing the program on different makes.

Bemer (1969) stated that he had not seen successful mechanical translation of machine language programs. Instead of looking for a solution to the existing problem, he advocated guidelines to be followed by assembly language programmers. These should prevent the loss of information that may occur. To achieve this he proposed attaching the source program "as a certification of a kind" to the object program. This proposal never found acceptance for two reasons. In the first place working storage at that time was limited. Therefore, it was not plausible to load the source code. Secondly, commercial producers of software that included the source code as part of their program packages incurred financial losses. Often their techniques were copied, without sharing in the yield.  

Wallis (1982) proposed a number of programmer guidelines for the ADA\textsuperscript{10} language to avoid future problems. Dahlstrand (1984) stressed the

\textsuperscript{9} The operating system of the Digital Equipment's PDP/8 computer was available in source code form. Many of its ideas are retraceable in the operating system developed for the CP/M machine and its successors.

\textsuperscript{10} ADA is not an acronym; it is the first name of lady Lovelace, a sentient lady that developed programs for Babbage's computer around 1850.
need for writing programs that are computer independent. This meant that the programmer should not rely on the specific features of one machine, but should take into account the rapid progress in computer technology that may influence the future performance of his program. He stated that programs could run on different machines only if they were prepared for it.

Lecarme and Pellisier Gart (1986) published an extensive study devoted to the problem. Like many earlier writers their main concern was how to avoid future problems, not to solve existing ones. Old programs remained outside their scope.

3.4.4 Views held by others

Sable (1969) noted that emulators had been used in the past with limited success or acceptance. He concentrated on the use of data in new applications and proposed the development of a data description language. Also, in 1969, Ward noted that the method of ‘transferability’

11, suggested by Strong (1969), had proved to be insignificant. According to Ward the lack of transferability made the upgrading of a computer system a traumatic ordeal. Barbe (1969) estimated that only two percent of existing software could be used on other machines. This minute percentage was not caused by a lack of effort on the part of the computer establishment. Existing technology made it practically impossible. There was an increasing awareness that the cost of programming was rising markedly, and that it could be reduced by facilitating access to existing programs. An important part of this cost increment was spent on manual reprogramming existing software, reliable, but extremely expensive.

The concept of using a universal computer language for higher language levels was also applied to assembly languages. As early as 1979, Fisher proposed a microprocessor assembly language standard. Later Fitz and Crockett (1986) proposed a universal assembly language. It was possible to devise such a language as most microprocessors, at that time, worked internally with registers of the same length

12. On the other hand Kaindl (1988) totally ignored the issues involved in using assembly programs or

Transferability: the possibility to move bit strings to another computer.

These registers were usually 8-bits.
object code on different machines.

Instead of contriving a universal programming language, another idea was to construct a universal operating system. Hamlet and Haralick (1980) proposed constructing, what they called, a kernel of routines that interfaces the peculiar operating system for each machine. This kernel's objective was to make the operating system of computers appear identical.

Construction of a universal operating system makes calls to the operating system suitable for different platforms but the remaining code is specific for each type computer and therefore needs adaptation to different platforms. Until recently this problem remained unattended. Attempts to develop a universal operating system are now being pursued to implement the same working environment on different platforms such as: computers that use the 68000 and 8086 processors and their successors\(^\text{13}\). These operating systems are designed to offer the user the same working environment, irrespective of the type computer. Higher-level language programs can be run without adaptation with these universal operating systems.

These are encouraging developments that may help alleviate future problems.

3.5 RE-ENGINEERING

Around 1990, the attention shifted from re-using whole programs to parts, called modules or routines. To quote Ulman (1991): “The great platonic ideal of portability still lingers somewhere just out of reach”. The term “portability” moved to the background. Others came into use such as: re-engineering, reverse engineering and re-use. (Cheng, 1998, Krueger, 1992). The source code programs came into focus again. The idea of setting standards for future software was reconsidered. According to Kesselmeier it is necessary to have specifications of the software, to be able to reuse it.

\(^{13}\) OMEN, later renamed OASIS, a multitasking operating system and Magic C are examples of this type of software. A more successful one is LINUX, a descendant of UNIX.
Attempts to transform software, available in machine code, will be further expounded. The major problem with machine code is to identify instructions and data and their sequential order. In the past two main approaches to solve this problem were followed. One uses code disassembly, called static analysis. The other relies on determining the event sequences and their order to discriminate between instructions and data, called dynamic analysis. The latter entails emulation of the program to be reconstructed and accruing information about it. Dynamic analysis is used in this thesis as a means to reconstruct the source code. Because only information about instructions executed is gathered, a number of instructions, like those that will only be executed when an error condition exists, will not be examined. Furthermore, code that is only included as comment will not be accessed. In this case the former has to be employed. In the next section disassembling will be investigated along with an introduction to dynamic analysis.

3.6.1 Code disassembly

Disassemblers have been in use for a long time and are still available. Adequate use depends on the information to be elicited from the user. Therefore, they can only be handled by the experienced. Disassemblers are normally used to generate source code when only machine code is available. Chae Woo Yoo (1985) used disassemblers in his efforts to, what he called, decompile. His work addressed only operating system software. Furthermore, he chose the method of static analysis and used source code programs whenever he tried to re-use applications. He also addressed code optimization.

Horspool and Marovac (1980) devoted a study to the problem of disassembly and called it detranslation. They first reconstructed a source program from its binary code applying the static approach. They did not use dynamic analysis because they felt it did not provide all the information required about the source program to be investigated. Therefore, their research was limited to code disassembly using static analysis. It is true that

---

14 Disassembler: a computer program that translates machine code into mnemonic code.
dynamic analysis does not provide all the information to reconstruct the source code, however, measures can be taken to retrieve data. These measures will be discussed in more detail later.

Although the approach of Horspool and Marovac is interesting it did not solve the reconstruction problem. Relying on static or dynamic analysis alone makes attaining a reliable solution impossible.

### 3.6.2 Dynamic analysis

Using dynamic analysis to solve the code reconstruction problem was the focus of May (1987). She describes a system that might have solved it. However, it did not materialize because her research work shifted to other, more urgent problems.

She categorized a group of instructions into modules. This notion was first introduced by Allen (1976). The first instruction of a module is a branch target, the last a branch instruction (see figure 4). Modules and dynamic analysis proved useful in solving the code reconstruction problem. Whether an old program is constructed of modules can be determined by dynamic analysis.

<table>
<thead>
<tr>
<th>description</th>
<th>label</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch instruction</td>
<td></td>
<td>branch to AAA ....</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..........................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..........................</td>
</tr>
<tr>
<td></td>
<td></td>
<td>..........................</td>
</tr>
<tr>
<td>branch target</td>
<td>AAA</td>
<td>..........................</td>
</tr>
<tr>
<td>branch instruction</td>
<td></td>
<td>branch to BBB</td>
</tr>
</tbody>
</table>

**figure 4 Module**

### 3.7 SUMMARY

The above discussion demonstrates that there has been interest in using software on different type machines throughout computer history.
Literature presented some useful notions that are applied in the present research. Previous work was based mainly on static analysis (Brown, 1977, Cowell, 1976, Thomas 1965). An alternative route uses event sequences. The determination of event sequences is very difficult when adhering to static analysis. Dynamic analysis was extensively used to reconstruct programs available in machine code. The following chapter will explore how it was used.