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.
4 TRANSFORMATION PROBLEMS

4.1 INTRODUCTION

This chapter describes a solution to a specific transformation problem. A decomposition of this problem will first be provided. This helps to clarify problems that have to be solved in succession. The ambiguity problem has to be solved before determining event sequences for the object computer. These events have to be identified before transformation of the code can take place. This must be done in such a way that proper event sequences are invoked. Three main routes can be followed to solve the code transformation problem and will be explored in this chapter. Of these the 'one-instruction-to-subroutine call method' proved the most promising. It maintains the relationship between the original and the transformed program. It also keeps instructions and data separated.

4.2 THE ORIGIN OF THE AMBIGUITY PROBLEM

In chapter one computers were defined as systems generating events. To generate these events physical processes have to be started and organized. In general, a physical system that endures over a relatively long period and can be stimulated to change its state can be used to generate events. Controlling such changes is done by instructions: sentences in a programming language. Conceptually, a program is composed of instructions. To execute on a computer they must be transformed to bit strings. Computer memory contains data as well as instructions for manipulating them. To determine whether bit strings are instructions or data is difficult when they are transferred to a computer that uses a different set of instructions. This problem would not exist if the source code used to obtain the machine code were available. All that needs to be done is to translate the instructions from one computer language to the other. When the source
4.2.1 Earlier efforts

The question may arise as to why the method to achieve transformation, proposed in this thesis, was not tried earlier. Different reasons for this can be provided. Existing literature held that reconstruction of programs available in machine code was unattainable. In computer literature the ambiguity problem is considered insurmountable. Horspool and Marovac and Yoo hold, that complete 'translation back' is impossible because instructions and data are stored in the same space, without any indication of what is an instruction and what is a datum1 (Opler, 1963).

In the past most researchers tried to solve the reconstruction problem using only static analysis. They all failed. The problem got the odium of being unsolvable. The prevailing attitude by the computer community during the early stages of technical development was not to pursue seemingly unattainable solutions. At that time main memory storage was scarce and processing speed was extremely slow compared to now. These were drawbacks in pursuing a solution to a problem that requires abundant main memory as well as secondary storage and an extensive amount of processing time.

Literature mentions one solution that used dynamic analysis. It was discussed in the preceding chapter. This promising development was stopped (May, 1987).

4.2.2 Problems to be solved

The main problem to be solved when reconstructing machine code programs is to identify the meaning of the bit strings that make up the program. Each type of computer attributes a different meaning to them. Bit strings are uniquely identified only in the context of the computer for which the program was written. In order to transform bit strings into source code a number of problems have to be solved. One is to determine which bit strings are instructions and which are data. This identification problem is addressed as the ambiguity problem.

1 This is not always the case as it depends on the type of computer used. For example, early NCR-computers provided for separate storage.
Another is to distinguish between operators and operands. This is because a number of instructions may consist of both. The problem is further compounded in that operands may be data to be processed or be used to influence the flow of the program. An operator is that part of the instruction which indicates what event has to take place. If the operator is 'move' the operand indicates what has to be moved. If the operator is 'branch' it indicates a target address. Therefore, it is essential to determine whether an operand influences the outcome of a calculation or the flow of the program. A distinction is made between operands referencing variables or constants pertaining to solutions of the problem, and those influencing the flow of the program. The former will be referred to as 'problem operands' and the latter 'control operands'. 'Add the value thousand to a certain variable' is an example of an instruction containing a problem operand. In this case 'thousand' is the problem operand. In the instruction 'branch to address thousand' the operand is a control operand.

When the operand consists of one or more bytes it is usually referred to as an immediate datum. When the operand points to a memory address it is an indirect address and the contents of that address is a datum (see figure 5).

<table>
<thead>
<tr>
<th>Hexadecimal Representation</th>
<th>Mnemonic Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>3E 41</td>
<td>LD A, 41</td>
</tr>
<tr>
<td>operator</td>
<td>operand</td>
</tr>
<tr>
<td>32 4A01</td>
<td>LD (014A), A</td>
</tr>
<tr>
<td>operator</td>
<td>operand</td>
</tr>
</tbody>
</table>

*figure 5 Operators and operands*

'LD A, 41' and 'LD (014A)' are instructions consisting of an operator and operand part. The operand '41' is an example of an immediate datum. The operand '4A01' points to the hexadecimal address '14A'. The con-
The contents of address 14A are a datum. ‘Add the contents of a certain address’ is an example of an instruction referring a problem datum. In the instruction ‘branch to the contents of address thousand’ a control datum is referred.

### 4.2.3 Problem and control operands

The value of problem operands and data has to be the same in original and transformed source code; otherwise the program will produce erroneous output. For control operands and data the situation is different. Its value is used to direct the flow of the program. If the continuation is to an address in the application there is no problem as that location in the object program can be computed. But, often the flow of the program is directed to functions of the operating system. In this case the address cannot be easily computed and requires more elaborate manipulation to achieve correct transformation.

To direct the program flow to the operating system a unique number is used. If the operating systems of both the source and object computer use the same conventions for using such functions there is no major problem. But, if they differ, the control operands and data have to be adapted in order to perform a similar function as on the source computer. Although operating systems of personal computers are similar in many aspects, there are differences that should be considered (see appendix A.3.3.4 for a table of function calls and their code). Because of these differences all system calls have to be checked and if necessary adapted. This guarantees that the same function will be performed on the object computer. When the source and object computer use a comparable operating system, the solution to the transformation problem is less complicated.

The problem becomes more complicated when a datum is used both as problem and control datum. This problem can be illustrated by the following example: Suppose a word processor continues depending on the character typed. To do so the American Standard Code for Information Interchange (ASCII) value of the character is added to the value of a fixed address. Then a jump is performed to the calculated address. The character entered then becomes a problem and a control datum. At the moment the instruction is executed, the computer does not "know", whether it concerns a problem or control datum. This only
becomes clear from the context of the instruction. Still another problem will arise if instructions of the source program are overwritten by other instructions when they are no longer needed. Other problems encountered may need user intervention.

To solve the code transformation problem different routes can be followed. They will be discussed in the next chapter. The next section discusses different routes that can be taken to solve the ambiguity problem.

4.3 SOLUTIONS TO THE AMBIGUITY PROBLEM: STATIC AND DYNAMIC ANALYSIS

Most efforts to overcome the ambiguity problem stem from static analysis of bit strings. An advantage of this method is that all bit strings, which make up the program, can be investigated. But often it fails to correctly distinguish between instructions and data. The first bit string in a program in machine language does not pose a problem. This is because it has to be an executable bit string. The operating system expects it to be so. A machine language program cannot start with a datum. After the first bit string has been investigated it becomes clear, depending on its nature, whether the next one is an instruction or a datum. However, usually instructions are not executed in their strict order. Programs containing branch instructions cause additional problems. The operand of a branch instruction points to a memory location where the next instruction to be executed can be found. Many branches are only executed when a specific condition is fulfilled. Whether the next location in memory contains an instruction or a datum can only be determined during execution of the program. This is because computers do not, normally, maintain a memory separation between instructions and data.

4.3.1 Static and dynamic analysis, advantages and disadvantages

MALIN uses both dynamic and static analysis to investigate the machine code program. Their advantages and disadvantages will be explained.

---

2 This is true for CP/M. Other operating systems may follow different conventions.
Dynamic analysis is a relatively simple method to determine whether bit strings are instructions or data and which data are used as a variable. It stores information of only executed instructions. This is a disadvantage because it fails to identify instructions that are not executed. It is possible that not all instructions of the old program (P) have been executed. Two reasons for this to occur are:

1. The program may contain branches for which the possible conditions are not fulfilled. The solution to this problem is to force their execution. Whether these instructions would ever be executed without such a manipulation is not important. At this stage it is only necessary to know whether the forced branch can be executed and whether it consists of instructions.

2. The program may contain superfluous instructions and/or data. When investigating a program to be transformed the method of dynamic analysis is applied first. This method determines the nature of the bit strings that make up the executable code. It is also used to identify instructions that will not be executed, unused data, such as a string of ASCII characters, inserted by its author as a comment etc.

Dynamic analysis eliminates the problem caused by the intermingling of instructions and data. It does this by storing information about the event sequences that take place during execution in a file. This information constitutes the program flow and the nature of instructions executed. The distinction is unambiguously made between instructions and data. A datum that is used as control and program datum is also clearly identified. An emulator running on the object computer accomplishes dynamic analysis.

4.3.2 Using dynamic analysis in MALIN

To perform dynamic analysis different emulators were developed. One, written in assembly language, not only emulates the program to be transformed but also the old working environment. This is done to guarantee that the behavior of the program is as near as possible to its behavior when running on the obsolete one. Another emulator, written in higher-level language, emulates the instructions of the old program. Calls to the operating system of the obsolete machine are converted into calls to that of the new. In contrast to the first emulator this one has the ability to execute instructions one by one. After each instruction executed the user
can inspect the registers and memory locations of M and change them, if required.

Both emulators are part of the transformation package. They store information about the source program during emulation in a file. If static analysis was used all bit strings were investigated only once. But with dynamic analysis, using an emulator, repetitive execution of the same bit strings cannot be avoided because programs usually contain loops. Loops are a number of instructions that might be executed several times until a certain condition is met. To determine whether the bit strings in the loop are instructions or data needs to be done only once. A provision has been made to ensure this.

The emulators determine whether an instruction belongs to the application program or to the operating system. If the user interacts with the computer, for example if manual input is required, functions of the operating system are used. Only instructions of the old program (P) need to be investigated. Instructions of the operating system need not, as they are not part of the code to be transformed. When the transformed program runs in the new environment its operating system is used. Calls to the old operating system need to be transformed to calls of the new. Handling of such calls is incorporated in the program that supports the functioning of the transformed program.

4.3.2.1 Inspecting the generated information

Dynamic analysis is used to determine, for every bit string, whether it is an instruction or a datum. By convention the first string of an executable file is an instruction\(^3\). If not, MALIN’s source code must be modified to insure that the program to be reconstructed starts at the correct address. This bit string is decoded by the emulator and uniquely defined. Furthermore, the type of instruction must be identified, such as: memory referencing and branches. Some instructions consist only of an operator and marked as such. If the instruction is not a branch instruction the next bit string must be an instruction. When the contents of an address are used as a datum this is also marked. All this information is stored in one file. A separate utility program distributes this information over separate

---

\(^3\) In computer literature called single stepping.

\(^4\) For CP/M programs this is a string of eight bits.
files to reduce access time. This has the added advantage that when there is a problem, during application of MALIN, the user can readily obtain information about the source program, as it is stored in plain ASCII code. One file contains the location of instructions executed, their source code, the hexadecimal value of operators and operands, if applicable. It contains the reconstructed source program. An example of part of it is as shown:

<table>
<thead>
<tr>
<th>address</th>
<th>hexadecimal</th>
<th>mnemonic code</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>3E41</td>
<td>LD A, 41</td>
</tr>
<tr>
<td>102</td>
<td>324A01</td>
<td>LD (014A), A</td>
</tr>
<tr>
<td>105</td>
<td>09</td>
<td>LD C, 9</td>
</tr>
<tr>
<td>105</td>
<td>113201</td>
<td>LD DE 0132</td>
</tr>
<tr>
<td>132</td>
<td>55</td>
<td>dc.b 'W'</td>
</tr>
<tr>
<td>133</td>
<td>68</td>
<td>dc.b 'h'</td>
</tr>
<tr>
<td>134</td>
<td>61</td>
<td>dc.b 'a'</td>
</tr>
<tr>
<td>135</td>
<td>54</td>
<td>dc.b 't'</td>
</tr>
</tbody>
</table>

The first column contains addresses, the second hexadecimal codes and the third their representation in assembly language. This example starts with four load instructions followed by a number of characters. The first instruction, LD, loads the accumulator with the hexadecimal value 41. The next instruction moves this value to the location with the hexadecimal address 14A. From location 132 - 135 the text 'What' is stored; 'dc.b' is an abbreviation of 'define constant as a byte'.

Another file contains information about a number of special situations, identified by MALIN that may need user attention. This file is generated to provide the user of MALIN with information that may be helpful to locate and remove errors that may be encountered. For example, when a branch is executed to the contents of a processor register incorrectly initialized.

---

5 An error refers to any computer instruction that results in other than what was intended.
4.3.2.2 Determining the type of branch instructions

Each time a branch instruction is decoded, using dynamic analysis, its address and the target address are stored in a separate file. An example of this file is as follows:

10A CALL 5
112 CALL 5
12C CALL 5
12F JP 0

If the necessary conditions to execute a branch are never present that branch will never be taken and the computer will always execute the next instruction in succession. This disadvantage of dynamic analysis was mentioned earlier. To be able to execute such branches, not only the branch address, but also information about which branch has been taken, must be stored. This is necessary to be able to force branches into execution. To make this task easier the information needed is generated by a utility program and stored in a separate file. This file contains information about branches used in the program. An example of such a file is shown below:

100 CD6205 CALL 0562H ;branch address 103H
106 CD8505 CALL 0585H ;branch address 109H
109 CDBD07 CALL 07BDH ;branch address 10CH
10C CD1101 CALL 0111H ;branch address 10FH

Every line in this file starts with the location of the branch. After the hexadecimal code of the instruction, for example CD6205, the type branch is stated followed by the target address. In this case, it is hexadecimal address 562. The last column contains the actual target address. If the same type conditional branch is encountered more than once at the same location and followed by a different target address then both branches have been executed. If not, the information about the branch executed is used in another utility of MALIN to generate code for the conditional branch not executed.

6 '10A' is not a typographical error. It is a hexadecimal number.
4.3.2.3 Rearranging the data in separate files

No new information was added to the single file generated by the emulator. By rearranging the data into separate files the user's insight may be enhanced. During the transformation process the user must be able to inspect the data gathered. Sometimes manual intervention is needed if MALIN took an obviously wrong decision, such as when a return instruction is used to return to the working environment, instead of returning from a subroutine instruction. To identify the location of this problem the generated files need to be inspected. Therefore, the data is stored in plain ASCII code in these separate files. They contain the mnemonic code of the instructions executed and the hexadecimal value of operands. During the transformation process they are used to determine which instructions have to be executed on the object computer. A message is generated and stored in a file if the emulator encounters a situation that requires user intervention. This file is maintained only for the user to decide whether the solution offered by MALIN is appropriate or a modification 'by hand' has to take place.

To obtain accurate transformation of a machine code program (P) to a source program (P') the generation of two files is necessary: one containing the reconstructed code and the other containing branches. The remaining files may be needed when the machine code program contains errors, superfluous instructions and/or data. When this occurs manual intervention may be needed.

4.4 USING STATIC ANALYSIS IN MALIN

When all branches in the original program (P) have been executed and the list of unused code, maintained by MALIN, still contains entries, they can be investigated by means of static analysis. To do so a dedicated utility generates a disassembly of this code. Usually this will be a message inserted by the programmer, such as a copyright message. The generated code can be appended as comment to the code already obtained by dynamic analysis.
4.5 THE PROBLEM OF DETERMINING EVENT SEQUENCES

To guarantee that each instruction, as it was executed on the source computer, has the same effect on the object computer, event sequences for these instructions have to be determined. Fortunately, they have already been developed and published; as such event sequences were needed in the development of emulators. However, event sequences that emulate the old operating system cannot be used because they let the new computer behave as if it were the old one. The determination of event sequences and some examples will be discussed further in chapter six.

4.6 THE CODE TRANSFORMATION PROBLEM

Once all bit strings of the program to be transformed have been investigated and their nature determined, the problem of transformation of the old code into code that can run on the object computer has to be solved. The reconstructed code is in the language of the source computer. This constitutes a problem when the contents of a register is used for a branch. The reason is that this code is an absolute address of a memory location. In such a case, transformation of the code requires special attention. Furthermore, the outcome of the transformation process can be at different programming language levels. It has to be decided which language level is the most appropriate to guarantee that correct executable code is obtained. Different routes to solve these problems were investigated. To help understand the advantages and disadvantages of these different routes some remarks and definitions are necessary.

4.6.1 Different routes to solve the code transformation problem

If a program written for machine M can be emulated on M', the event sequences of M' have the same result as when the program was executed on M. To transform machine code program (P) instructions to the language of the object computer M', first an interpretation as discussed above has to take place. MALIN ensures that for every program P a program P' is generated in such a way that the event sequences in M produce equivalent output for M'.
To obtain $P'$ that fulfills this requirement three different routes were explored:

**Route one (transformation of one instruction to many)**

Every instruction from the source program ($P$) is transformed to several instructions for the object computer ($M'$). Each series of instructions that unambiguously form the transformation of an instruction of $P$ is labeled. The same applies to its data. These labels are used by MALIN to make correct relations between them.

There are several subroutines to deal with the difference in the registers of the source and object computer. Thus, an unavoidable overhead is connected with the execution of transformed programs. The large number of instructions generated, far more than the original program contained, is a disadvantage. It makes the output of MALIN difficult to recognize as a transformation of the original. Furthermore, elaborate bookkeeping is needed to manage all code and data.

**Route two (transformation in high-level language statements)**

Every instruction in the source program is transformed to a statement in a higher language. Executable code is generated by an appropriate compiler or executed by an interpreter. Computer literature recommends the use of higher-level languages to solve problems with the aid of a computer. But, for code transformation, it is not suitable. Programs, except trivial ones, contain branches. If such a program needs to be changed branching can cause serious problems if indiscriminately used. Jumping in and out of subroutines is an example of bad programming habits. It is often encountered in old programs. Higher-level languages either forbid or discourage such programming technique. Using such constructs in a higher language leads to error messages and rejection of code generation. Most higher-level languages have qualifying power: their compilers accept the source program *only* if it obeys strict programming rules. This route of generating higher-level language statements was not used because of these problems.

**Route three (transformation of instructions into subroutine calls)**

Every instruction in the source program is transformed in a subroutine-call for the object computer program. A strict separation between instructions, operands and data is maintained. The area originally used by the source computer to execute the program is treated as a region containing
only operands and data of the object program. The disadvantage of this method, having to call a subroutine each time an instruction of the source program is executed, is relatively insignificant, when compared to its advantages. The high speed of today’s computers far exceeds those of yesteryear. Therefore, using many subroutine calls has little negative impact on execution. Its main advantage is a one-to-one relation between the source and object program instructions. Meaningful names of subroutines to be executed enable the user of MALIN to easily understand which instructions the source computer originally executed. The problem of running into data is avoided by making a separation between instructions to be executed and data they manipulate. Thus, the possibility of a program crash has been minimized, because one part of working storage will contain only instructions to be executed, and the other the data to be manipulated. This separation has been highly recommended in computer literature. However, it is seldom found in practice. By maintaining a separate data memory each datum has to be located. This results in an increase in the number of instructions to be executed. Safety has its price.

4.6.2 Choosing a route to solve the code transformation problem

These three routes represent the main strategies. The major differences between the program, as it might have existed in source code form, and the transformed code made route one difficult to use. The second route would be preferable to the first. Higher language statements need less expertise to be interpreted by the user than code on lower programming language levels. But, its earlier mentioned problems proved to be too much of a hindrance. One of the reasons the third route was chosen was because the output closely approximates the source code.

Other routes such as: direct code generation during emulation or interpreting the code of the old program during execution and directing system calls to the operating system of the object computer were considered but rejected. They did not fulfill the objective that is: the transformed code must not only be readable by the object computer but also by the user. This objective must be met to ensure that the user can adapt the code to meet his specific needs.
The three routes mentioned above were investigated and tested. The results are discussed in the following chapters.

The route chosen is restricted to machine code programs and to a specific combination of source and object computers. The package can be adapted to other configurations. In this case specific data about these computers have to be incorporated.

In this thesis dynamic analysis is applied to effect code transformation. This method may create unexpected variation requiring user intervention. To facilitate this intervention MALIN produces output that can be inspected by the human user. Some literature considers any solution solved by a computer that requires user intervention as inadequate. In a subsequent chapter this attitude will be further explored.

4.7 SUMMARY

How to reconstruct lost source code is limited to machine code programs for a specific obsolete machine. It is not practical to develop a computer program that would solve the problem for every type computer. The wide differences in register use and methods of addressing would entail extensive effort and be cost prohibitive. Only the software at the application level was considered for code reconstruction. No attempt has been made to reconstruct operating system software. However, this is possible, if desired.

The problem of identifying instructions and data is important and was discussed. To solve the ambiguity problem dynamic analysis of bit strings that formed the original program was necessary. The advantage of using separate files to store information that becomes available during dynamic analysis was also discussed. Attention was given to previous efforts to solve the code transformation problem and why they failed. Various routes to solve it were discussed and evaluated. The one-instruction-to-subroutine call route was the most promising and was used in the design of MALIN. This will be expounded further in chapter six.

The next chapter discusses the development of MALIN, problems encountered and tools developed to overcome them.