6 CODE RECONSTRUCTION AND GENERATION

6.1 INTRODUCTION

In previous chapters the transformation problem has been decomposed into two major ones. The first, the ambiguity problem, was discussed in chapter four. Its solution orders the bit strings of an old program (P) so its type can be unambiguously determined. The second is the transformation of the information gathered into source code.

This chapter examines:
1 How the dynamic analysis method was implemented to solve the ambiguity problem.
2 The transformation process.
3 Problems that became apparent during the development of MALIN. These were discussed in the foregoing chapter and called first order errors. Their solution will be discussed in more detail in this chapter.
4 The structure of MALIN.
5 Some important modules such as the development of an emulator and exmulator.
6 Transformation examples.

6.2 THE DESIGN OF MALIN

Instead of writing one big monolithic program that could solve the ambiguity and the transformation problem in one run, modules were written to solve specific problems. The collection of these modules makes up MALIN. The advantages of writing modules are that:
1 Adaptations to modules are easier to effectuate than to a monolithic program.
2 MALIN can run on computers with a modest amount of memory. The package has also been tested on computers with 512K main memory.
3 Information gathered can be inspected after the execution of each module.

How these modules are related to each other is clarified in pseudo higher-level language statements as follows:\footnote{A more detailed description is contained in the appendix.}:

```
emulate CP/M-program
   convert code in Z80-machine code
   determine_not_executed_branches
until all branches are executed

transform generated code
   assemble
   test and remove errors from transformed CP/M program
until all errors are removed
```

MALIN performs two main programming loops. The first has to be repeated until all branches have been executed. This guarantees that all executable code has been examined. The first module in this loop converts the binary code of the old program (P) into assembly code for the obsolete computer. The second module in the first loop ensures that both the ‘then’ and ‘else’ part of the ‘if’ statements have been executed.

The second loop transforms the generated code into code for the object computer (M'). This has to be repeated until the user has removed all errors. The first module in the second loop assembles the transformed code with the aid of a standard assembler of the object computer. The program can be considered faithfully reconstructed when the event sequences generated by the transformed code have the same effect on M' as they had on M.

The transformed code contains a number of subroutine calls that evoke the event sequences. A runtime system has been developed as part of MALIN that contains these subroutines. Furthermore, it handles conversion of system calls and executes them.
6.2.1 The development of MALIN

Step one

A utility had to be written to determine whether a bit string is used as an instruction or datum. There are programs available that can follow the program flow of a machine language program, to inspect and/or change the contents of the registers of the processor, to state addresses where the program has to stop (called breakpoints) etc. Such programs are called debuggers or tracers. They decode and execute every instruction of the program under consideration. An emulator performs almost identically as a debugger. It follows the program flow as executed on the old computer and executes a different set of instructions. In contrast a debugger normally executes only the computer’s instructions.

For MALIN two emulators were developed. The first has been written in machine code of the object computer. It functions in the working environment of the old one, which is simulated on the object computer. Fortunately, it was not necessary to write such an emulator totally anew. Different emulators for emulating code for the 8080 and Z80 processors, which are used by CP/M machines, are available in the public domain. Its source code has been published (Cathey, 1986). This code was used to develop the MALIN emulator. The code published by Cathey was well suited as it contains a very clever mechanism to trap system calls to the old operating system².

This emulator is intended primarily to determine the program flow in the old working environment. It proved to be valuable in researching old programs but could not be used to trace every possible branch. Examples of conditions that are difficult to generate are ‘Hard Disk Faulty’ and ‘Out of Memory’ with state of the art computers. Therefore, a second emulator that could cope with these types of condition was developed and used in step two.

Step two

An emulator, written in the higher-level language BASIC, made it possible to inspect the registers of the emulated source computer and to change the contents of its working storage. It also emulates the old program but not the code of the old operating system. Calls to the operating system of M are transformed in calls to M’. Development of this emulator

² Others used this code to develop an emulator that is available in the public domain.
consisted of transforming a tracer for programs using the CP/M operating system into one running on the object computer. Blok had developed such a tracer in assembly language for the CP/M operating system. There was no need to develop this emulator in assembly language because no time-critical processes were required. Tracing of CP/M programs on another computer is not time critical.

Step two was decomposed into several sub steps. Each of these was implemented as the program developed. In this process the first sub step was to design an instruction decoder. Next, a test program was written to execute the CP/M processor instructions. For every instruction of the processor a procedure had to be written. The third sub step was to add instructions to these procedures. They store, when executed, their type and the order in which the old program is executed in a file.

As this emulator is written in a higher language, users can easily modify it. It can, for example, be adapted for other combinations of source and object computer.

Step three
To determine whether a program (P) written for a CP/M-system could run on the object computer (M') an emulator was developed. It is called emulator because it emulates the instructions of the source program (P) and executes its function calls on the object computer (M'). This emulator executes all calls to the operating system of the source computer (M) on the object computer (M') and emulates the instructions of the source program (P).

Step four
If the program can be executed by the emulator on M', the next step, code generation of P, can be taken. This consists of determining the event sequences that have the same result on the source as on the object computer. To guarantee that on the object computer the same results are achieved as on the source, some registers of the source computer, like the accumulator, are simulated. One of the utilities of MALIN is used to ensure that the contents of the registers of the source and object computers are the same during execution. When a difference is encountered this

3 Blok, Optimization of an operating system, in an unpublished document
utility generates a message. It states where differences occur between the registers.

6.2.2 Other utilities of MALIN

To guarantee that the transformed code (P') is identical to the original code (P), the code reconstructed by MALIN is assembled. A special CP/M assembler was designed to perform this function. The code, generated by it, must be identical to P otherwise an error exists and must be corrected. For each instruction of the source computer an event sequence has been determined in such a way that the effect the instruction has on one or more registers of the source computer is the same for the object. Test programs have been written to verify this.

Some instructions of the source computer evoke event sequences that consist of the execution of two instructions on the object computer. For example, the load register instruction of the source computer needs only an ‘add quick’ and ‘move byte’ instruction on the object computer to be executed as shown below.

\[
\begin{align*}
\text{ldc:} & \quad \text{addq.l} \ #1, A1 \quad (\text{ldc} = \text{load contents of 'C'\ register})
\text{move.b} \ (A1)+, \text{regc(A5)} \\
& \quad \text{rts}
\end{align*}
\]

The above instructions are invoked by the following code in the transformed source program:

\[
\begin{align*}
\text{move.l} & \ #\text{ldc}, A0 \\
\text{jsr} & \ (A0)
\end{align*}
\]

The load register instructions can be found on every computer. But, for a number of instructions the event sequence on M' consisted of two or more instructions. Page 117 of the appendix contains an overview of instructions of the source computer and number of required instructions needed on the object computer.
6.3 SOLVING BRANCH PROBLEMS

Branch instructions can cause first order problems. Most originate when the address is contained in a register. During emulation the branch addresses becomes known. Each such address is labeled. One of the modules of MALIN constructs a table of branch addresses and their labels. This table is also assembled so that it can be inspected during execution. The address where the program has to continue can be readily ascertained by the runtime system of MALIN:

```
call branch
0107 CD6A03 CALL 036AH
36A branch address
used CALL-branch at 107

return branch
0327 C9 RET
6D0 branch address
used RET-branch at 327

jr branch
032F 2803 JR Z,03H
331 branch address
334 branch address
used JR-branch at 32F

jr branch
03FD 38F5 JR C,F5H
3FF branch address
not used JR-branch at 3FD
```

The module ‘jphl’, that is a routine of the runtime system, does the preparatory work. It ensures that the pointer into the data memory points to the continuation address. The module ‘jphlit’ converts the contents of the simulated ‘hl’ register into the address where the program has to continue. See the following example:
Conditional branches are handled by a test instruction. As a result of the test the branch is executed or the program continues with the next instruction. Below is an example of the transformation of a conditional branch instruction.

```
move.l #jphlt,A0
jsr (A0)
```

6.4 TESTING TRANSFORMED CODE

When the transformed code (that is the P’code) has become available it is good practice to test it thoroughly. This means testing under different conditions. For example, a word processor can direct its output to a disk or printer. In such cases both possibilities should be tested. To facilitate locating errors a special run time system has been developed. It tests for every transformed instruction whether or not the original program flow is followed. To do so the original code that is contained in the data memory is tested. If a difference is found the run time system prints an error message. It also includes the address of the error. This makes it relatively simple to rectify. A trace flag is maintained. When it is set to one it enables other important information to be stored in a file. An example of the load
RE-ANIMATION OF COMPUTER PROGRAMS

'bc' register routine, of the runtime system, is shown below with the testing instructions indicated by a rectangle.

<table>
<thead>
<tr>
<th>ldbc_n</th>
<th>move.b</th>
<th>(pseudodpc)+, D0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmp.b</td>
<td>#01, D0</td>
<td></td>
</tr>
<tr>
<td>beq.s</td>
<td>gldbc_n</td>
<td></td>
</tr>
<tr>
<td>move.l</td>
<td>#error, A0</td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td>(A0)</td>
<td></td>
</tr>
<tr>
<td>gldbc_n</td>
<td>move.l</td>
<td>#inext, A0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jmp</td>
<td>(A0)</td>
<td></td>
</tr>
<tr>
<td>tst</td>
<td>trcflg</td>
<td></td>
</tr>
<tr>
<td>bmi.s</td>
<td>ldbc_n</td>
<td></td>
</tr>
<tr>
<td>move.l</td>
<td>#trac16b, A0</td>
<td></td>
</tr>
<tr>
<td>jsr</td>
<td>(A0)</td>
<td></td>
</tr>
</tbody>
</table>

The code for the load register instruction is 01. This code is tested in the first two lines of the routine. If no difference is found the branch to the label 'gldnc_n' is taken, otherwise a jump to the error routine is performed. If the trace flag is set to one the contents of the 16-bit register is stored. Then the instructions to load the register are executed. The runtime system, without testing possibilities, directly executes the instructions following the label 'ldbc_nn'.

6.5 SOLVING AD HOC PROBLEMS

The solution to the code reconstruction problem could not be realized until other related problems were solved. All calls to the operating system had to be adapted. The more the operating systems of the source and object computer differ, the more code transformation has to take place. It is a fortunate circumstance that the same manufacturer had developed the operating systems of the source and object computer used.
One of the rules for calls to the operating system of the source computer is that the ‘c’ register contains the function number. The transformed code looks like this:

```
move.l #ldc,A0
jsr (A0)
move.l #func,A0
jsr (A0)
```

The first two lines of code in this example guarantee that the simulated register ‘c’ contains the function number of the system call that has to be invoked. The module ‘func’ manipulates the code it finds in the simulated ‘c’ register into a call to the operating system of the object computer. Nevertheless, the following problem occurred. To guarantee that programs developed for the source computer could run under different versions of the operating system and could use different I/O devices, the developers of the operating system provided for strict programming rules. These are not always followed. Often programmers use undocumented features. For example, they write code that uses jumps direct into the operating system. Such jumps, when encountered, had to be managed by writing a special routine for the run time system. User intervention may be needed when a program contains a non-standard jump.

Sometimes programmers also use the operand part of an instruction as the target address of branches. In such cases the target address was transformed into a labeled entry point. When the normal instruction order was followed the entry point was circumvented, otherwise a branch to the label was executed.

Another problem that may be encountered is the possible encountering of errors. In chapter five this problem was thoroughly addressed. Once identified, the order of the error has to be determined through use of a debugger. If it is a first order error user intervention is needed. Second order errors must be solved by user intervention. As the source code of MALIN is available, the competent user can resolve these. Errors in the compiler, assembler or operating system of the object computer, are third order errors and beyond the scope of this thesis.
6.6 COMPARISON OF ORIGINAL AND TRANSFORMED CODE

In the following example three lines of a reconstructed program and the transformed code are printed.

<table>
<thead>
<tr>
<th>Reconstructed code</th>
<th>Transformed code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD C, 9</td>
<td>move.l #1dc,A0</td>
</tr>
<tr>
<td>LD DE, ASK</td>
<td>jsr (A0)</td>
</tr>
<tr>
<td>CALL 5</td>
<td>move.l #1dde_n, A0</td>
</tr>
<tr>
<td></td>
<td>jsr (A0)</td>
</tr>
<tr>
<td></td>
<td>move.l #func, A0</td>
</tr>
<tr>
<td></td>
<td>jsr (A0)</td>
</tr>
</tbody>
</table>

The first line of the original code directs the function number of the display routine, which is '9', to be loaded in register 'C'. This routine expects the address of the string to be displayed in the register pair 'DE'. The third line is a call to the operating system.

Each line of the transformed code calls a subroutine that generates the events necessary to effect what originally happened in the source computer. These routines have been given names that come as close as possible to the mnemonics of the instructions in the reconstructed code (see appendix A.13).

A comparison of the reconstructed and the transformed code reveals that the transformed one contains twice the number of instructions as the original. This is because the address of each subroutine to be executed is loaded in a register, after which the 'jsr' instruction performs the actual call. The total number of instructions is the sum of the move and jsr instructions plus the instructions of the subroutines.

The number of instructions executed, if the code is transformed, is far less than under emulation. Therefore, the execution time is reduced. There are also other factors that save time. If the code is transformed, calls to the original operating system are, in most programs, significantly reduced. At regular intervals most CP/M programs test whether the user prefers to terminate the program. As the operating system of M' performs the same function, this test is performed twice during emulation. Under transformation this double testing does not apply.
6.7 DIFFERENCES IN INSTRUCTION SETS

The complexity of the instruction set of the source computer strongly determines how complicated the transformation package will be. The memory locations of the CP/M machine, the source computer, are eight bits wide. The instructions of the object computer can operate on bit strings of different lengths, usually in multiples of eight. Some of the registers of the source computer are 16 bits wide. The high 8 bits as well as the 8 low bits of these registers can be addressed separately. When a part of a 16-bit register has to be loaded, a move byte instruction of the object computer must be used. If 16 bits have to be loaded, a move word instruction must be used. For every source computer instruction an equivalent one for the object computer is normally present. If not, more instructions are needed to obtain the same result.

6.8 EXAMPLES OF CODE RECONSTRUCTION

The next paragraph describes the reconstruction of two programs: A disassembler and ‘Word Star’. The latter, a word processor, was originally developed for CP/M computers.

6.8.1 The disassembler

At first it may seem odd to reconstruct a disassembler since the package produces a disassembly of the program to be reconstructed. But, by reconstructing the old disassembler a comparison can be made with the output produced by MALIN. This comparison provides an insight into the problems encountered designing a disassembler and shows how the output of an old disassembler is different from the one incorporated in MALIN. Disassembling a program with an old type disassembler was accomplished in steps. Some disassemblers and compilers do not immediately produce usable output. They first produce an intermediate code to be transformed by a separate program. These steps are called passes. Normally, no user intervention is needed between passes. If no errors are detected the computer can perform the next pass. A static disassembler produces code for each bit string examined. The user must make a distinction between instructions and data. The next step consists of running
the disassembler with the same object code as input. The user must state
the beginning and ending addresses of data. Old disassemblers that per­
formed static research were unable to distinguish instructions that could
not be executed under any conditions. The human agent had to perform
tedious research to determine the flow of the program. In general, what
formerly had to be done manually has become a mechanical task.
Technological progress, in conjunction with advanced programming
techniques, has made this possible. A modern disassembler supports the
connection with the past by its ability to reconstruct the original source
code and makes it possible to access hidden information.

6.8.2 Word Star

As an example of how the connection with the past can be retained Word
Star was reconstructed. Although there is a deluge of word processors it
may be worthwhile, if only for historical reasons, to reconstruct such an
obsolete program. Before giving a detailed description of the problems
encountered in reconstructing this software and how they were solved
some introductory notes about Word Star should be made.

The main memory or working storage of earlier computers was limited.
This created a problem for software developers. For example, the standard
CP/M computer was equipped with only 32K memory. If a program
needed more memory it had to be split up. When a program exceeded
main memory the part that performed initialization was brought into
memory. Seldom used functions were stored separately on the disk. They
were brought into memory only if the program called for them. This part
of the program, called an ‘overlay’, overlaid functions in memory not
immediately required for continuation of the program. This was a com­
pliation for MALIN that had to be overcome. Some programs, by using
different overlays, could circumvent the main memory limitations of
these earlier computers.

Currently, personal computers, with practically unlimited main
memory, no longer need to use this overlay technique. Main memory has
become inexpensive. The need to economize is no longer applicable.
This abundance of main memory can cause problems for the user. Some
word processing programs assume that computers have more than ample
main memory. So, when a document is to be input that exceeds the main
memory space allocated, the program simply states "not enough memory". Word processing programs that were developed for CP/M computers did not have this deficiency because a technique, comparable to the overlay method, was used to manage large amounts of data. Often human and mechanical agents run into trouble because they are unaware of imposed constraints.

At the time Word Star, one of the first word processors, was developed, graphic possibilities were limited. The operating system of earlier computers seldom contained graphic routines. They became available later. Word Star had a graphical appearance by simulating graphic routines that redraw the screen depending upon the human agent's actions. Earlier computers could not randomly access the screen. Each time a change had to be made, different from the cursor position, the screen had to be totally rewritten.

When Word Star was introduced the mouse did not exist. It was necessary to manipulate the arrow keys to move the cursor. The typist had to avoid cursor manipulation as much as possible and concentrate on the correct entering of text. Prior to the development of menus, special key combinations were required to invoke certain functions. Also, the title of the file had to be typed in. With the advent of the mouse and menus the user need only to select and click. Some continue to prefer the textual invocation of functions rather than manipulating the mouse. The latter requires different actions: opening a menu, selecting the desired function and clicking. By entering text the user can concentrate on one action: typing.

Programs such as Word Star also used menus. Because of hardware constraints menus had limited possibilities. Word Star displays a menu that can be removed by the user. It displays the names of the functions and the necessary key combinations required to invoke them. In some instances, the user has to perform many keystrokes.

Word Star provides the user a program that functions even with limited working storage (the maximum working storage of CP/M machines was 64 K). Furthermore, a beginner can use a word processor that constantly displays functions and the necessary keystrokes. Word Star's success can be attributed to its user friendliness. However, it had its constraints. This led to the development and success of Word Perfect that overcame

---

4 For example, Tempus, a word processor for the ATARI computer, has this drawback.
many of Word Star's limitations. It is not as user friendly as Word Star. To use Word Perfect requires an extensive learning process. It is widely used on IBM PC's and its clones. Initially, these were not equipped with a mouse. To help users select the right keystroke combinations function templates were devised. They are positioned around the function keys. Microsoft's Windows permits IBM PC's to use a mouse and menus. This eliminated the need for templates.

Reconstructing Word Star provides the possibility to compare the old with the new. This may be of interest to programmers, information historians and others.

6.9 RECAPITULATION

In this chapter the steps taken in the development of MALIN, were discussed. Some utilities to test whether the transformed code conforms to the original were presented. Attention was given to potential problems and their solution. Although the solution to code reconstruction, provided by MALIN, is limited to a specific combination of computers it is applicable to other configurations. Some examples were provided to illustrate MALIN's capabilities.

5 In Word Star the template can be constantly displayed on the screen. Therefore, it is not necessary to memorize the instructions or to look at the keyboard. The user can concentrate on typing without being distracted.