

*Master Thesis in
Software Engineering
Thesis no: MSE-2002:17
June 2002*



Designing an object-oriented decompiler

Decompilation support for Interactive Disassembler Pro

David Eriksson

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE-372 25 Ronneby

This thesis is submitted to the Department of Software Engineering and Computer Science at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:

Author:

David Eriksson

Address: Folkparksvägen 14:32, 372 40 Ronneby

E-mail: david@2good.nu

University advisor:

Lars Lundberg

Department of Software Engineering and Computer Science

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
SE - 372 25 Ronneby
Sweden

Internet : www.ipd.bth.se
Phone : +46 457 38 50 00
Fax : +46 457 271 25

Abstract

Decompilation, or reverse compilation, takes a computer program and produces high-level code that works like the original source code. This makes it easier to understand a computer program when source code is not available. However, there are very few tools for decompilation available today. This report describes the design and implementation of Desquirr, a decompilation plug-in for Interactive Disassembler Pro. Desquirr has an object-oriented design and performs basic decompilation of programs running on Intel x86 processors.

The low-level analysis uses knowledge about specialized compiler constructs, called idioms, to perform a more accurate decompilation. Desquirr implements data flow analysis, meaning the conversion from primitive machine code instructions into code in a high-level language. The major part of the data flow analysis is the Register Copy Propagation which builds high-level expressions from primitive instructions. Control flow analysis, meaning to restore high-level language constructs such as if/else and for loops, is not implemented.

A high level representation of a piece of machine code contains the same information as an assembly language representation of the same machine code, but in a format that is easier to comprehend. Symbols such as '*' and '+' are used in high-level language expressions, compared to instructions such as "mul" and "add" in assembly language. Two small test cases which compares decompiled code with assembly language shows promising results in reducing the amount of information needed to comprehend a program.

Keywords: decompilation, reverse engineering, program transformation

Contents

1	Introduction	3
1.1	Reverse compilation	3
1.2	Goal and objectives	4
1.3	Outline for remaining chapters	4
2	Method	5
2.1	Make a basic decompiler for 32-bit 80386 machine code	5
2.2	Make the decompiler object-oriented	6
3	Data flow analysis	7
3.1	Overview	7
3.2	Static compared to dynamic analysis	8
3.3	Idioms	8
3.3.1	C calling convention	8
3.3.2	Memcpy an unknown number of bytes	9
3.3.3	Memcpy a constant number of bytes	9
3.3.4	Compare and set boolean	10
3.3.5	The question-mark-colon operator	10
4	Design of the decompiler	11
4.1	Data structures	11
4.2	Data structure example	11
4.3	Function calls	12
4.4	Analysis class	12
5	Results	13
5.1	Decompiling the Fibonacci calculation	13
5.2	Decompiling the palindrome test	13
5.3	Object-orientation of decompiler	13
6	Discussion	22
7	Conclusions	23
A	Desquirr class reference	26
A.1	Node class hierarchy	26
A.2	Instruction class hierarchy	26
A.3	Expression class hierarchy	27

List of Figures

1.1	Relations between high-level language, assembly language and machine code.	3
3.1	Restore stack with one POP for each function call	8
3.2	Restore stack with one ADD ESP for each function call	8
3.3	Restore stack for two calls with one ADD ESP for each function call	9
3.4	Memcpy an unknown number of bytes	9
3.5	Memcpy a constant number of bytes	9
3.6	Compare and set boolean; assembly language version	10
3.7	Compare and set boolean; translated version	10
3.8	Question-mark-colon operator	10
4.1	Object tree for <code>add eax, 2</code>	12
5.1	C source for Fibonacci calculation	14
5.2	Decompiled Fibonacci calculation	15
5.3	Disassembly of Fibonacci main function	16
5.4	Disassembly of Fibonacci calculation function	17
5.5	C source for palindrome test	18
5.6	Decompiled palindrome test	19
5.7	Disassembly of palindrome main function	20
5.8	Disassembly of palindrome test function	21
A.1	Class diagram for the Node class hierarchy	26
A.2	Class diagram for the Instruction class hierarchy	27
A.3	Class diagram for the Expression class hierarchy	28
A.4	Class diagram for the Location part of the Expression class hierarchy	28

Chapter 1

Introduction

1.1 Reverse compilation

There are cases when you need source code for a computer program but the source code is not available. For example, you need to perform maintenance on some old company software and can not find the source code. You may also want to port an application from one platform to another. If you have been attacked by an unknown virus or trojan, you definitely want to examine what damage it has caused [1], [2]. It is also possible that you have source code available, but you need to verify that your compiler produced correct machine code for your really safety-critical system. The most efficient support tool for the tasks listed above is a decompiler.

The translation from machine code to a high-level language is called *reverse compilation* or *decompilation*. A high-level language is a programming language which provides some level of abstraction above assembly language. It is translated to machine code by a compiler. Machine code is a set of instructions coded so that the CPU can use it directly without further translation.

Decompilation may also be compared to disassembling, where the latter means to translate from machine code to assembly language. Assembly language is a textual representation of machine code. It is translated to machine code with an assembler. See figure 1.1 for a graphical view of the relations between high-level language, assembly language and machine code. Decompilation may also be applied to Java byte-code and Visual Basic P-code, but only decompilation of machine code is discussed further in this thesis.

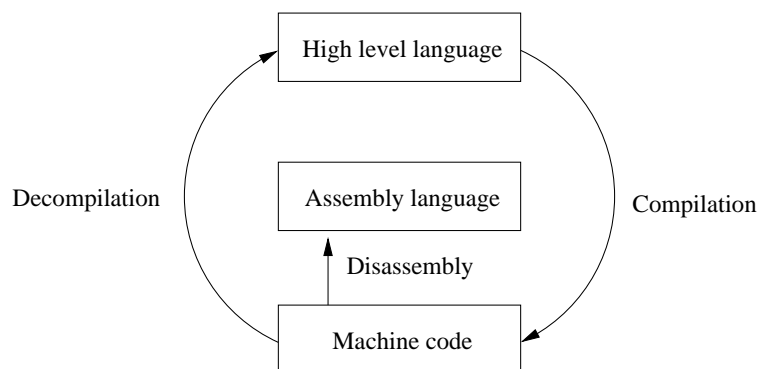


Figure 1.1: Relations between high-level language, assembly language and machine code.

The foundations of reverse compilation comes from compilation, especially the optimization part of a compiler.

Machine code is relatively easy translated into assembly language with a disassembler. There are sim-

ple disassemblers such as Objdump in GNU Binutils [3] and advanced disassemblers such as Interactive Disassembler (IDA) Pro [4]. A simple disassembler starts at address zero and disassembles everything sequentially. An advanced disassembler begins disassembling at the program starting location and continues by looking at the execution flow. The assembly language is a long list of primitive instructions, one per line. In a high-level language such as C, many primitive instructions are joined together into an expression on (most often) only a single line. This suggests that the greatest benefit of reverse compilation is the reduced effort required to comprehend a computer program that is only available in machine language. By converting machine code to a high-level language (such as C) the number of lines of information may be reduced significantly and the context of the program becomes much clearer.

Decompilers have existed since the early 1960s, but reverse compilation is still a relatively small area of academic research. It is related to the reverse engineering and reengineering disciplines. One reason for the limited research in this area may be legal issues, as a reverse compilation tool may be abused. More about legal issues can be found in [5]; they are not discussed further in this thesis. Today, the only previously available decompiler for Intel x86 machine code is limited to 16-bit Intel 286 machine code compiled for MS-DOS [6].

In my personal experience, a more modern implementation of an application should use an object-oriented development process. When developing complex software, such as decompilers, there are high requirements of flexibility and extendability. An object oriented development process supports a more structured approach to solving the problem. The use of object oriented techniques when developing decompilers will allow more complex and flexible systems in the future.

This report is intended for people with knowledge of Intel 80386 assembler and basic compiler techniques. Object-oriented design skills are also useful.

The latest information about Desquirr is available at <http://www.2good.com/software/desquirr/>.

1.2 Goal and objectives

- Make a basic decompiler for 16- and 32-bit Intel 386 machine code
- Make the decompiler design object-oriented

1.3 Outline for remaining chapters

The next chapter will discuss the method for making the decompiler and using an object-oriented design. It also includes how the results will be evaluated. The following two chapters describes how the decompilation in Desquirr works and gives an overview of the object-oriented compiler design. Next, the results from using the decompiler are presented. The results are followed by a discussion and the report ends with my conclusions. There is also an appendix with details about the object-oriented design.

Chapter 2

Method

2.1 Make a basic decompiler for 32-bit 80386 machine code

The initial idea for this project came from finding the dcc decompiler [6] written in the beginning of the 90's by Cristina Cifuentes as part of her Ph.D. thesis [7]. Dcc is a very well working decompiler, but with one major shortcoming: it is only capable of decompiling 16-bit applications for MS-DOS. Because of this limitation, the **Desquirr** project was started. The major goal with Desquirr is to provide decompilation of both 16- and 32-bit computer programs compiled for any operating system running on Intel x86.

Due to the short time available for this thesis, I had to limit the scope of the decompiler. To be able to concentrate on the actual decompilation I decided to use the commercial tool Interactive Disassembler Pro [4] as a foundation for the decompiler.

Reverse compilation can be divided in two major parts: data flow analysis and control flow analysis. I decided to concentrate on data flow analysis, thus not implementing control flow analysis. The reasons for this choice was that data flow analysis makes a better job at reduced the amount of information in machine code and that control flow analysis methods had been studied previously [8].

- Data flow analysis covers the conversion from primitive machine code instructions to expressions in high-level language.
- Control flow analysis convert conditional and non-conditional jump instructions into high-level language control constructs such as if/else, switch/case for-loops and while-loops.

Interactive Disassembler (IDA) Pro from DataRescue is a commercial tool for disassembly of binary programs. It runs on the MS Windows and OS/2 platforms but is capable of loading and disassembling binaries for many different platforms. The first version of IDA was released in 1991. IDA Pro is currently on version 4.21 and now supports many common CPUs (such as the Intel x86 series) and virtual machines (such as Java).

IDA Pro also has a plugin interface which allows developers to add custom processing of loaded machine code. This plugin interface was used to create the Desquirr decompiler. The benefit of using IDA Pro as a base for developing a decompiler is that it already handles loading of binary files and identification of library functions. This saves a lot of work, but creates a dependency on a commercial entity.

The effectiveness of the Desquirr decompiler plugin will be measured by counting non-empty lines in code listings.¹ One line represents one instruction. This kind of comparison may seem rather coarse, but an high-level instruction is easier to understand than the sequence of assembly language instructions representing the high-level instruction. The reason for this is that a high level representation of a piece of machine code contains the same information as an assembly language representation of the same machine code, but in a format that is easier to comprehend. Symbols such as '*' and '+' are used in high-level language expressions, compared to separate instructions such as "mul" and "add" in assembly language.

¹Exact line number calculation: `grep -cv '^$' filename`

2.2 Make the decompiler object-oriented

The design goal for the Desquirr plugin is to make it object-oriented. To solve this, object oriented design of compiler optimization was studied. I found a report by Johnson et al [9] describing the RTL System, an object-oriented framework for code optimization. RTL is an abbreviation for Register Transfer Language.

Cifuentes dcc decompiler is written with very little object-orientation in mind, so hardly no design ideas can be borrowed from this application. The RTL System, however, provides class hierarchies for basic blocks (called "flow nodes"), instructions (called "register transfers") and expressions. These will be adapted to suit decompilation.

The use of object-oriented techniques will be evaluated based on a reasoning of whether the RTL System was suitable for adaption to decompilation.

Chapter 3

Data flow analysis

3.1 Overview

The Decompilation in Desquirr is performed like this, step by step:

1. Perform **low-level analysis**. Desquirr starts by making a list of machine code instructions for the current function. This information is gathered from IDA Pro. This list is then iterated to find **idioms** and convert low-level instructions into high-level instructions and expressions. See section 3.3 for details about idioms.
2. Separate list of instructions into **basic blocks**. These are important elements in compiler and decompiler theory [10]. A basic block is a sequence of instructions inside a function. A basic block ends with a conditional jump, unconditional jump, return from a function or a "fall through" if the next basic block is the target of a jump.
3. Calculate uses and definitions for all instructions. A register is said to be **Defined** when a value is assigned to this register. A register is said to be **Used** when it is referenced but not modified by an instruction.
4. Perform live register analysis. **Live registers** denote processor registers that are defined in one basic block and used in another basic block. The live registers are useful to detect whether an optimization is safe or not. **LiveIn** for a basic block is a list of Live registers on the entry to this basic block, and **LiveOut** is a list of the registers that are Live at the exit from the basic block. The algorithm for calculating the LiveIn and LiveOut lists is provided by Cifuentes [7].
5. Find **DU-chains**. DU is an abbreviation of Definition-Use and DU-chains are an important aid in both compiler optimization and decompilation [10]. A DU-chain is used to connect the definition of a processor register with the uses of this definition. A DU-chain is created by taking each definition of a register in an instruction and find every use of the register before it is redefined or the basic block ends. Cifuentes [7] describes DU-chains in more detail.
6. **Register Copy Propagation**. This is the first part of the data flow analysis. This requires DU-chains and Live registers and is a simple but very effective algorithm. Register copy propagation means that when a definition only has one use, we replace the use with the definition. This algorithm is also provided by Cifuentes [7].
7. **Find function call parameters**. This part of the data flow analysis takes stack push instructions and convert them into function call parameters.
8. **Code generation**. Print the basic blocks in decompiled format. The format of decompiled code is made to look like C code.

If control flow analysis had been implemented, this would have occurred after data flow analysis and before code generation. It is also worth noting that no step in the data flow analysis adds more instruction that it removes. This guarantees that the number of instructions in the decompiled version of a program is always less than or equal to the number of instructions in the machine code.

3.2 Static compared to dynamic analysis

If we only read machine code from a binary and output a listing with decompiled code, we have performed a static analysis. This is unfortunately not bullet-proof; we may need user intervention to perform successful decompilation.

The Intel 80x86 and Pentium series of CPUs are types of von Neumann machines. This means that code and data are represented the same way and share the same address space. This makes it impossible to make a perfect distinction between code and data in a program. In Desquirt, I use three approaches to handle this problem:

- Primarily Desquirt lets IDA Pro decide what is code and what is data. This analysis handles most cases.
- If the plugin encounters unknown bytes within a function, it asks IDA Pro to try to generate code from those bytes.
- Due to the interactive nature of IDA Pro, the user is able to manually decide what is code and what is data.

3.3 Idioms

An idiom is a special way of performing a certain operation. An example of a simple idiom is to use `xor eax, eax` to clear register `eax` and not use `mov eax, 0`. The number of clock cycles required is two for both versions on the 80386 CPU. The former instruction requires two bytes of memory, but the latter requires five bytes, making it clear that this is a size optimization only. The knowledge of idioms is vital to be able to perform adequate decompilation.

Cifuentes defines an idiom as *a sequence of instructions that has a logical meaning which cannot be derived from the individual instructions* [7]. She describes certain idioms that she found during her work with the dcc decompiler. This section aims at adding to her collection of idioms.

3.3.1 C calling convention

Two idioms for the C Calling Convention are described by Cifuentes [7]. In the C Calling Convention, parameters are pushed on the stack by the caller before a call and the caller is responsible for restoring the stack when the call returns. This normally looks like this for multiple function calls:

	<code>push eax</code>	<code>; push second parameter</code>
	<code>push ebx</code>	<code>; push first parameter</code>
<code>push eax</code>	<code>; push parameter</code>	<code>call function_c</code>
<code>call function_a</code>	<code>; call function A</code>	<code>add esp, 8</code>
<code>pop ecx</code>	<code>; restore stack pointer</code>	<code>push eax</code>
<code>push eax</code>	<code>; push parameter</code>	<code>push ebx</code>
<code>call function_b</code>	<code>; call function B</code>	<code>call function_d</code>
<code>pop ecx</code>	<code>; restore stack pointer</code>	<code>add esp, 8</code>
		<code>; restore stack for call D</code>

Figure 3.1: Restore stack with one POP for each function call

Figure 3.2: Restore stack with one ADD ESP for each function call

However, an optimized version of this was found in code generated by the gcc compiler, where the stack was not cleared immediately after each call:

```
push  eax          ; push second parameter
push  ebx          ; push first parameter
call  function_c   ; call function C
push  eax          ; push second parameter
push  ebx          ; push first parameter
call  function_d   ; call function D
add   esp, 10h     ; restore stack for both function call C and D
```

Figure 3.3: Restore stack for two calls with one ADD ESP for each function call

The first two constructions make it very simple to deduce the number of parameters to a function. The latter, however, makes an unaware decompiler to wrongly assume that the function call before `add esp` takes all the parameters restored from the stack and that the previous function call has no parameters. Desquirm attempts to guess the correct number of parameters, but most correct guess of stack parameter count comes from analyzing the called functions. Desquirm is capable of reading such information from IDA Pro, but does not attempt to generate this information on its own.

3.3.2 Memcpy an unknown number of bytes

This sequence of instructions is actually an inlined memcpy function or a C++ copy constructor. Desquirm translate it to `memcpy(edi, esi, ecx)`, meaning to copy `ecx` bytes from offset `esi` to offset `edi`.

```
mov   ecx, ecx    ; save ecx in eax
shr   ecx, 2      ; shift ecx two bits right (divide by four)
repe movsd        ; move four bytes, repeat ecx times
mov   ecx, eax    ; restore ecx from eax
and   ecx, 3      ; mask out the lower two bits
repe movsb        ; move one byte, repeat ecx times
```

Figure 3.4: Memcpy an unknown number of bytes

3.3.3 Memcpy a constant number of bytes

One more optimized version of memcpy for a constant number of bytes was found in code generated by the gcc compiler. The constant specifies a number of 32-bit words (four bytes) that will be moved, and if the area is not a multiple of four bytes, a `movsw` may be added to move two more bytes, and a `movsb` for one additional byte. Desquirm translate it to `memcpy(edi, esi, numeric-literal)`, where `numeric-literal` is `4*CONSTANT` bytes, plus two bytes if `movsw` is present and plus one byte if `movsb` is present. For example, to copy 255 bytes, `CONSTANT` is 63 and both `movsw` and `movsb` are present in the idiom.

```
mov   ecx, CONSTANT
repe movsd
movsw                ; optional
movsb                ; optional
```

Figure 3.5: Memcpy a constant number of bytes

3.3.4 Compare and set boolean

This is a way of setting a register to zero or one depending on the result of a comparison. Example code follows:

```
xor    edx, edx
cmp    esi, ecx
setz   dl                    edx = (esi == ecx)
```

Figure 3.6: Compare and set boolean; assembly language version

Figure 3.7: Compare and set boolean; translated version

3.3.5 The question-mark-colon operator

The special assembly language instruction sequence below is used to represent the C/C++ question-mark-colon operator. Figure 3.8 an example of this idiom.

What happens in the two first instructions is that if the `eax` register is initially set to zero, it is still zero. If `eax` is non-zero from start, it will be set to `0xffffffff`. The last instruction will have no effect on `eax` if `eax` is zero, but will set `eax` to the value of the second operand otherwise. The if-statement version of the expression is included below for the sake of clearness.

<pre>neg eax sbb eax, eax and eax, 8h</pre>	<pre>eax = (eax ? 8 : 0)</pre>	<pre>if (eax) eax = 8; else eax = 0;</pre>
Assembly version	Translated version	If-statement version

Figure 3.8: Question-mark-colon operator

Chapter 4

Design of the decompiler

Desquirr currently consists of a little more than 5000 lines of C++ code, not counting empty lines or lines beginning with comments.¹

4.1 Data structures

The decompiler uses several data structures, where the most important are nodes, instructions and expressions. These are described in this section.

The design of this decompiler plugin is heavily inspired by the RTL System. The RTL System uses the `RTLFlowNode` class and its subclasses to represent nodes in a control flow graph. Although this plugin does not provide control flow graph analysis, certain algorithms need to know the number of successors to a node, and therefore I provide not only a `Node` class but also a class hierarchy (figure A.1) similar to that of `RTLFlowNode`.

The RegisterTransfer class hierarchy in RTL is represented by the Instruction class hierarchy (figure A.2) in the decompiler plugin. One important difference between the decompiler plugin and RTL is that a Call is a type of expression and not a separate instruction such as Assignment, Jump and ConditionalJump.

The Expression class hierarchy (figures A.3 and A.4) in the decompiler plugin is inspired by the RTL-Expression class hierarchy in the RTL System but differs in several ways. For example, a ternary (three-operand) expression has been added to support the C/C++ question-mark-colon operator as described in section 3.3.5.

Details about data structures in Desquirr are available in Appendix A.

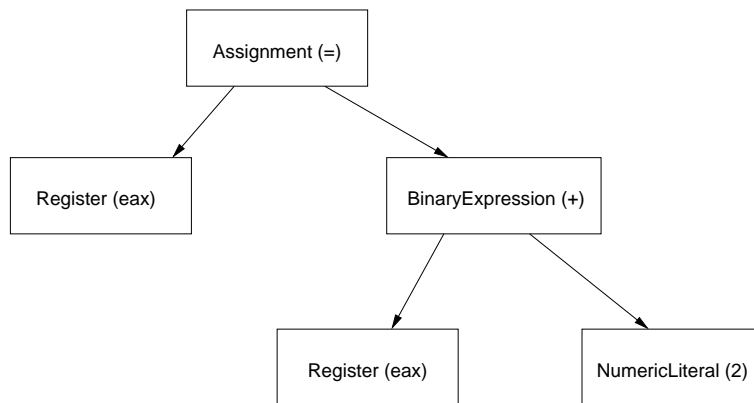
4.2 Data structure example

This section will describe how an instruction is translated into objects. Let us start with the primitive machine code instruction `add eax, 2`. This means "eax = eax + 2" and therefore is an assignment instruction. We will create an object of type **Assignment**, which has the **Instruction** type as base class. Before we can create the Assignment object, we also need to create the **Expression** objects for the operands.

An assignment takes two operands, source and destination. The destination operand is simple; a **Register** expression that stores `eax`. The second operand is not that simple, we have to create a **BinaryExpression** with "+" as operator and the operands as subexpressions.

The first operand in the **BinaryExpression** is a **Register** expression, just like the result, and the second is a **NumericLiteral** with value 2. The resulting object hierarchy is available in figure 4.1.

¹ Exact line number calculation: `cat *.cpp *.hpp | grep -vE '^\s|^\s*/'`

Figure 4.1: Object tree for `add eax, 2`

4.3 Function calls

Dcc [6] has a special basic block ending with a `CALL` instruction. This is not the case in Desquirr, because Desquirr currently only perform register copy propagation within basic blocks, and more basic blocks would give less possibilities for register copy propagation.

The RTL System and dcc both have a special high-level instruction for function calls. In Desquirr, a function call is only a type of expression. The reason for this difference is that I did not want to treat function calls that do not return values differently from those that do return values. A function that does not return a value simply makes an assignment to dummy destination.

4.4 Analysis class

A special base class called `Analysis` was introduced in Desquirr. The purpose of this class is to provide an easy traversal of `Node` and `Instruction` vectors while adding or removing instructions. To perform an analysis, the `Analysis` class is inherited in a new class (such as `LowLevelAnalysis`) and appropriate methods for this analysis are overloaded.

Chapter 5

Results

5.1 Decompiling the Fibonacci calculation

First, figure 5.1 shows C source code for Fibonacci calculation, courtesy of Cifuentes [7], with 24 non-empty lines. This C source code was compiled into machine code with Turbo C++ version 3.0 (a compiler from 1992). Second, figure 5.2 shows the decompiled version of the machine code, with 27 non-empty lines.

The figures 5.3 and 5.4 show disassembly of the machine code for comparison, with a total of 84 non-empty lines.

In this example, the decompiled code was 32% of the disassembly, or about one third in line count. Note that the disassembly is poorly optimized, as seen in the Fibonacci function, where there is one "jmp short loc 10318" that is never reached and a superfluous "jmp short \$+2".

The decompiled code was only three lines longer than the C source code, but is missing header file includes, curly braces and variable declarations.

5.2 Decompiling the palindrome test

A palindrome is a word or sentence that reads the same both forward and backward. The C program in figure 5.5 tests if a string is a palindrome. The C source code was compiled into a 32-bit Microsoft Windows console application with Borland C++ 5.5 (from 2000). The disassembly is shown in figures 5.7 and 5.8. The C source code has 35 non-empty lines and the machine code disassembly has 82 non-empty lines. The decompiled version (figure 5.6) has 31 non-empty lines, which is 37% of the disassembly. This is also four lines shorter than the C source code, but the decompiled version is of course missing curly braces and variable declarations.

5.3 Object-orientation of decompiler

The design of the decompiler was successfully inspired by the RTL System although the design was not mirrored. There are two major reasons for not just implementing the RTL System design. First, the RTL System is made for a compiler and not a decompiler. Second, the RTL System uses Static Single Assignment (SSA) form. For more information about SSA, see the discussion in chapter 6.

```
#include <stdio.h>

int main()
{
    int i, numtimes, number;
    unsigned value, fib();

    printf("Input number of iterations: ");
    scanf ("%d", &numtimes);
    for (i = 1; i <= numtimes; i++)
    {
        printf ("Input number: ");
        scanf ("%d", &number);
        value = fib(number);
        printf("fibonacci(%d) = %u\n", number, value);
    }
    exit(0);
}

unsigned fib(x)    /* compute fibonacci number recursively */
int x;
{
    if (x > 2)
        return (fib(x - 1) + fib(x - 2));
    else
        return (1);
}
```

Figure 5.1: C source for Fibonacci calculation

```
sub_10291:
    _printf("Input number of iterations: ");
    ax = _scanf("%d", & var_2);
    si = 1;
    goto loc_102DD;

loc_102AF:
    _printf("Input number: ");
    _scanf("%d", & var_4);
    var_6 = sub_102EB(var_4);
    ax = _printf("fibonacci(%d) = %u\n", var_4, var_6);
    si = si + 1;

loc_102DD:
    if (si <= var_2) goto loc_102AF;

    _exit(0);
    return ax;

sub_102EB:
    if (arg_0 <= 2) goto loc_10313;

    dx = sub_102EB(arg_0 - 1);
    ax = sub_102EB(arg_0 + 0xffff);
    ax = dx + ax;
    goto loc_10318;

    goto loc_10318;

loc_10313:
    ax = 1;
    goto loc_10318;

loc_10318:
    return ax;
```

Figure 5.2: Decompiled Fibonacci calculation

```

sub_10291  proc near          ; CODE XREF: start+155

var_6      = word ptr -6
var_4      = word ptr -4
var_2      = word ptr -2

        enter 6, 0
        push si
        push offset aInputNumberOfI ; format
        call _printf                                10
        pop cx
        lea ax, [bp+var_2]
        push ax
        push offset aD      ; format
        call _scanf
        add sp, 4
        mov si, 1
        jmp short loc_102DD

loc_102AF:                                     20
        ; CODE XREF: sub_10291+4F
        push offset aInputNumber ; format
        call _printf
        pop cx
        lea ax, [bp+var_4]
        push ax
        push offset aD_0      ; format
        call _scanf
        add sp, 4
        push [bp+var_4]        30
        call sub_102EB
        pop cx
        mov [bp+var_6], ax
        push [bp+var_6]
        push [bp+var_4]
        push offset aFibonacciDU ; format
        call _printf
        add sp, 6
        inc si

loc_102DD:                                     40
        ; CODE XREF: sub_10291+1C
        cmp si, [bp+var_2]
        jle loc_102AF
        push 0 ; status
        call _exit

        pop cx
        pop si
        leave
        retn                                50
sub_10291  endp ; sp = -0Ch

```

Figure 5.3: Disassembly of Fibonacci main function

```

sub_102EB  proc near          ; CODE XREF: sub_10291+35
                                         ; sub_102EB+10 ...

arg_0      = word ptr 4

          push  bp
          mov  bp, sp
          push si
          mov  si, [bp+arg_0]
          cmp  si, 2
          jle  loc_10313
          mov  ax, si
          dec  ax
          push ax
          call sub_102EB
          pop  cx
          push ax
          mov  ax, si
          add  ax, 0FFFEh
          push ax
          call sub_102EB
          pop  cx
          pop  dx
          add  dx, ax
          mov  ax, dx
          jmp  short loc_10318

          jmp  short loc_10318

loc_10313:                                     ; CODE XREF: sub_102EB+A
          mov  ax, 1
          jmp  short $+2

loc_10318:                                     ; CODE XREF: sub_102EB+24
                                         ; sub_102EB+26
          pop  si
          pop  bp
          retn

sub_102EB  endp

aInputNumberOfI db 'Input number of iterations: ',0 ; DATA XREF: sub_10291+5
aD             db '%d',0 ; DATA XREF: sub_10291+10
aInputNumber  db 'Input number: ',0 ; DATA XREF: sub_10291+1E
aD_0          db '%d',0 ; DATA XREF: sub_10291+29
aFibonacciDU db ' fibonacci(%d) = %u',0Ah,0 ; DATA XREF: sub_10291+42

```

Figure 5.4: Disassembly of Fibonacci calculation function

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>

void rev(char* source, char* destination)
{
    char* tmp = destination + strlen(source);
    for (*tmp-- = 0; *source; *tmp-- = *source++)
        ;
}
10

int main(int argc, char**argv)
{
    char* original = NULL;
    char* reverse = NULL;

    if (argc < 2)
    {
        original = "nitalarbralatin";
    }
    else
    {
        original = argv[1];
    }
    reverse = malloc(strlen(original)+1);
    rev(original, reverse);

    if (0 == strcmp(original, reverse))
    {
        printf("%s is a palindrome\n", original);
    }
    else
    {
        printf("Try again!\n");
    }

    free(reverse);
    return 0;
}
20
30
40
```

Figure 5.5: C source for palindrome test

```
sub_401150:
    bx = arg_0;
    ax = _strlen(bx) + arg_4;
    * ax = 0;
    ax = ax - 1;
    goto loc_40116E;

loc_401167:
    dl = * bx;
    bx = bx + 1;
    ax = ax - 1;
    * (ax + 1) = dl;
    10

loc_40116E:
    if ((* bx) != 0) goto loc_401167;

    return ax;

_main:
    if (argc >= 2) goto loc_401188;
    20

    bx = "nitalarbralatin";
    goto loc_40118E;

loc_401188:
    bx = * (argv + 4);

loc_40118E:
    si = _malloc(_strlen(bx) + 1);
    sub_401150(bx, si);
    if (_strcmp(bx, si) != 0) goto loc_4011C7;
    30

    _printf("%s is a palindrome\n", bx);
    goto loc_4011D2;

loc_4011C7:
    _printf("Try again!\n");

loc_4011D2:
    _free(si);
    return 0;
    40
```

Figure 5.6: Decompiled palindrome test

```

_main      proc near          ; DATA XREF: .data:0040A0D0

argc       = dword ptr 8
argv       = dword ptr 0Ch
envp       = dword ptr 10h

          push  ebp
          mov  ebp, esp
          push ebx
          push esi
          cmp  [ebp+argc], 2
          jge short loc_401188
          mov  ebx, offset aNitalarbralati ; "nitalarbralatin"
          jmp  short loc_40118E
loc_401188:                ; CODE XREF: _main+9
          mov  eax, [ebp+argv]
          mov  ebx, [eax+4]
loc_40118E:                ; CODE XREF: _main+10
          push ebx          ; s
          call _strlen
          pop  ecx
          inc  eax
          push eax          ; size
          call _malloc
          pop  ecx
          mov  esi, eax
          push esi          ; int
          push ebx          ; s
          call sub_401150
          add  esp, 8
          push esi          ; s2
          push ebx          ; s1
          call _strcmp
          add  esp, 8
          test eax, eax
          jnz short loc_4011C7
          push ebx
          push offset aSIsAPalindrome ; format
          call _printf
          add  esp, 8
          jmp  short loc_4011D2
loc_4011C7:                ; CODE XREF: _main+3F
          push offset aTryAgain ; format
          call _printf
          pop  ecx
loc_4011D2:                ; CODE XREF: _main+4F
          push esi          ; block
          call _free
          pop  ecx
          xor  eax, eax
          pop  esi
          pop  ebx
          pop  ebp
          retn
_main      endp

```

Figure 5.7: Disassembly of palindrome main function

```

sub_401150  proc near           ; CODE XREF: _main+2B

arg_0      = dword ptr 8
arg_4      = dword ptr 0Ch

        push    ebp
        mov     ebp, esp
        push   ebx
        mov     ebx, [ebp+arg_0]
        push   ebx           ; s
                                                10
        call   _strlen
        pop     ecx
        add     eax, [ebp+arg_4]
        mov     byte ptr [eax], 0
        dec     eax
        jmp     short loc_40116E

loc_401167:                ; CODE XREF: sub_401150+21
                                                20
        mov     dl, [ebx]
        inc     ebx
        dec     eax
        mov     [eax+1], dl

loc_40116E:                ; CODE XREF: sub_401150+15
        cmp     byte ptr [ebx], 0
        jnz     short loc_401167
        pop     ebx
        pop     ebp
        retn
                                                30
sub_401150  endp

aNitalarbralati db 'nitalarbralatin',0 ; DATA XREF: _main+B
aSIsAPalindrome db '%s is a palindrome',0Ah,0 ; DATA XREF: _main+42
aTryAgain      db 'Try again!',0Ah,0 ; DATA XREF: _main+51

```

Figure 5.8: Disassembly of palindrome test function

Chapter 6

Discussion

The primary source of knowledge that the Desquirr decompiler is based on is the Ph.D thesis [7] by Cristina Cifuentes. Her thesis is a major work in the field of reverse compilation. As part of the Ph.D thesis a decompiler called dcc was created [6]. Dcc provides a valuable reference regarding tool support for reverse compilation.

Cifuentes has continued to work in the field of reverse compilation, including work on the University of Queensland Binary Translator (UQBT) [11]. UQBT is capable of taking a binary for Solaris running on a SPARC CPU and translate it into a binary for Linux running on a Pentium CPU. The UQBT project will be used in a newborn open source decompiler project called Boomerang [12]. Unfortunately, the source code to UQBT has not yet been released to the public and has therefore not been studied as part of this thesis. An interesting future task would be to compare Desquirr with UQBT.

In order to support certain compiler optimizations we need to adapt the decompiler for each optimization. To implement this support we have to test the decompiler on many programs and see how the compiler has optimized the machine code and how well the decompiler handles the optimization. C++ templates and optimizations such as function inlining and loop unrolling may be impossible, or at least very complicated, to detect. Therefore it would be equally difficult to decompile such these optimizations in a way that is similar to the original code.

When converting machine code to high-level code we need to detect and propagate data types through the program. This is not implemented in Desquirr. Single integers are easy to handle, but arrays, `struct` and `class` data types are much harder. There is a risk that different high-level constructions are compiled into a single construction in machine code. This makes it harder to correctly reverse the compilation.

The lack of data type handling in Desquirr is also the reason that Desquirr does not produce compilable C code. The format of decompiled code is made to look like C code, although it is not compilable without editing.

Another problem seen during decompilation is that idioms some times becomes "scrambled" due to compiler optimizations, probably because of pipelining. The current implementation of idiom handling in Desquirr requires that instructions in the idiom are sequential.

IDA Pro is capable of finding and describing certain switch/case idioms. Therefore, switch and case instructions were added to Desquirr and the switch/case information provided by IDA Pro was used to create switch and case statements in the decompiled code. Structure data types declared by IDA Pro are also used by Desquirr, but support for enumerations is not yet implemented.

The RTL System uses Static Single Assignment (SSA) form, but Desquirr does not. The meaning of SSA form is that a variable is only assigned once and so called ϕ -instructions (phi-instructions) are used to join variables at the beginning of a basic block. The purpose is to make optimizations easier. For an introduction to SSA and references for further reading, see section 8.11 in *Advanced Compiler Design and Implementation* [13]. Using SSA form in Desquirr is a possible future project.

The addition of control flow analysis would make Desquirr an even more useful decompiler. The Analysis abstract base class may have to be refactored in order to support control flow analysis and other kinds of analyzes not currently implemented in Desquirr. We may also need new methods on other data structures. Further development may warrant the use of the Visitor pattern from [14] for the data structures in Desquirr.

Chapter 7

Conclusions

The use of Interactive Disassembler Pro as a foundation for this decompiler was a good decision and allowed me to concentrate on the reverse compilation parts. However, the dependency on a commercial entity may be undesirable.

With the aid of a few simple algorithms we can significantly improve the possibilities to understand a program that is only available in machine code, but it would be a monstrous task to make a decompiler capable of perfectly decompiling most applications.

Desquirr was capable of decompiling two small example programs into about as many lines of decompiled code as lines of C code in the original program. The decompiled code does not include curly braces or variable declarations and only has goto statements instead of loop structures.

No step in the decompilation process adds more instruction lines than it removes. This means that the number of lines of decompiled code is guaranteed to be less than or equal to the number of machine code lines.

The framework created for this plugin can serve as a base for future work in making a more sophisticated decompiler. A more advanced compiler may still utilize the same data structures with few modifications.

References

- [1] Cristina Cifuentes, Trent Waddington, and Mike Van Emmerik.
Computer security analysis through decompilation and high-level debugging.
In *Proceedings of the Working Conference on Reverse Engineering, Workshop on Decompilation Techniques*, pages 375–380. IEEE Press, October 2001.
- [2] Eugene H. Spafford.
The internet worm program: An analysis.
Technical Report Purdue Technical Report CSD-TR-823, West Lafayette, IN 47907-2004, 1988.
- [3] GNU Binutils.
<http://www.gnu.org/software/binutils/binutils.html>.
- [4] IDA Pro.
<http://www.datarescue.com/idabase/>.
- [5] Cristina Cifuentes.
Reverse engineering and the computing profession.
IEEE Computer, pages 168, 166–167, December 2001.
- [6] The dcc decompiler.
<http://www.it.uq.edu.au/groups/csm-old/dcc.html>.
- [7] Cristina Cifuentes.
Reverse Compilation Techniques.
PhD thesis, Queensland University of Technology, 1994.
ftp://ftp.it.uq.edu.au/pub/CSM/dcc/decompilation_thesis.ps.gz.
- [8] D. Simon.
Structuring assembly programs.
Honours thesis, The University of Queensland, Department of Computer Science and Electrical Engineering, 1997.
- [9] Ralph E. Johnson, Carl McConnell, and J. Michael Lake.
The RTL system: A framework for code optimization.
Technical Report 1698, Urbana, Illinois, 1991.
- [10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman.
Compilers: Principles, Techniques and Tools.
Addison-Wesley Publishing Company, 1986.
- [11] UQBT - a resourceable and retargetable binary translator.
<http://www.itee.uq.edu.au/~csmweb/uqbt.html>.
- [12] The Boomerang decompiler.
<http://boomerang.sourceforge.net/>.
- [13] Steven S. Muchnick.
Advanced Compiler Design and Implementation.
Morgan Kaufmann Publishers, 1997.
- [14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York,
NY, 1995.

Appendix A

Desquirr class reference

Class diagrams in this section are provided in UML notation.

A.1 Node class hierarchy

The different node types corresponds do different actions in a control flow graph.

Node Abstract base class for all nodes.

OneWayNode Abstract parent class for nodes with one out-edge.

TwoWayNode Abstract parent class for nodes with two out-edges.

ReturnNode Represents a basic block that ends with a Return instruction.

FallThroughNode Represents a basic block where execution continues on the adjacent following node.

JumpNode A basic block that ends with a Jump instruction.

ConditionalJumpNode This basic block ends with a ConditionalJump instruction.

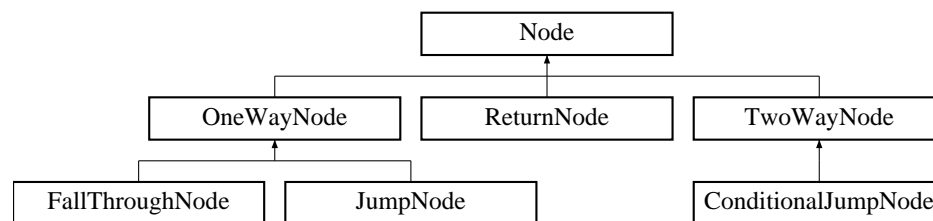


Figure A.1: Class diagram for the Node class hierarchy

A.2 Instruction class hierarchy

The instruction class hierarchy (figure A.2) is divided into two major groups: UnaryInstruction and BinaryInstruction. These represent instructions with one or two parameters, respectively.

Instruction Abstract base class for all instructions.

Label Class that represents a label, that is a destination of a Jump. This instruction has no operands.

LowLevel This instruction represents a low-level instructions as it is represented by IDA Pro. It has to be processed into other instructions before code generation.

UnaryInstruction Abstract parent class for single-operand instructions.

BinaryInstruction Abstract parent class for dual-operand instructions.

Assignment Assigns from any expression to a primitive expression.

ConditionalJump Jumps to a location if a condition is met.

Jump Unconditional jump to a location.

Push Push an operand on the stack. All push instructions should be eliminated before code generation.

Pop Pop an operand from the stack. All pop instructions should be eliminated before code generation.

Return Return a value from a function.

Switch Represents the header of a switch statement. The Switch instruction is present because IDA Pro is capable of detecting certain types of switch constructions.

Case This is a case label that is part of a switch statement.

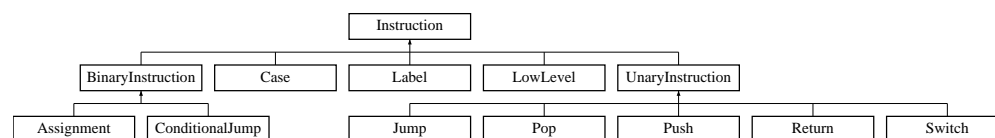


Figure A.2: Class diagram for the Instruction class hierarchy

A.3 Expression class hierarchy

Different types of expressions are represented by different classes.

Expression Abstract base class for all expressions.

UnaryExpression Class representing single-operand expressions.

BinaryExpression Class representing dual-operand expressions.

TernaryExpression Class representing three-operand expressions. In practice this is used for the C/C++ question-mark-colon operator.

Call Expression containing a function call.

Dummy An expression that has no value. Used a placeholder during analysis.

Location An abstract class for expressions that represent named locations.

Global A named global location.

StackVariable A named local variable.

NumericLiteral A primitive expression holding a number.

StringLiteral A primitive expression holding a string value.

Register A primitive expression representing a CPU register.

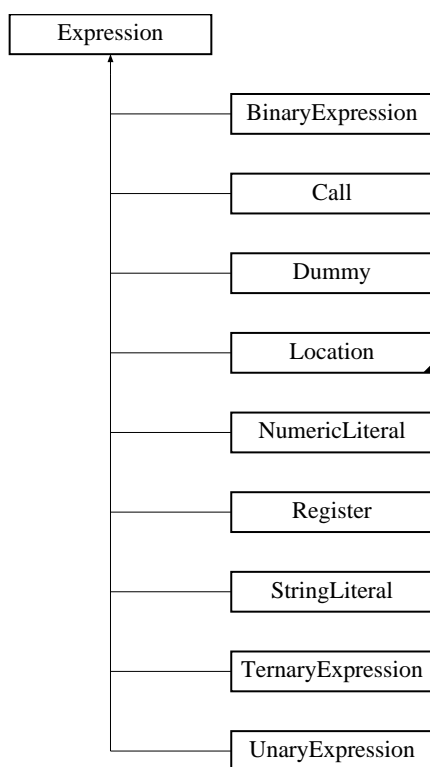


Figure A.3: Class diagram for the Expression class hierarchy

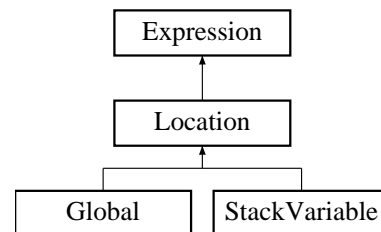


Figure A.4: Class diagram for the Location part of the Expression class hierarchy